University of Alberta

Retargetable Arithmetic Architecture for Low-Power Baseband DSP Supporting the Design for Reusability Methodology

by

Hongfan Wang

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59896-9

Canada

University of Alberta

Library Release Form

Name of Author: Hongfan Wang

Title of Thesis: Retargetable Arithmetic Architecture for Low-Power Baseband DSP Supporting the Design for Reusability Methodology

Degree: Master of Science

Year this Degree granted: 2000

Permission is hereby granted to the University of Alberta to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly of scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Hongfan Wang

Hongfan Wang
8907-112 St., Apt. 02B
Edmonton, Alberta
Canada T6G 2C5

Date: October 2, 2000

Abstract

Given the prevalence and sophistication of wireless communication devices, contemporary digital designers are increasingly challenged while developing various baseband digital signal processing (DSP) systems with better adaptability, higher throughput and less power consumption. Much attention is being paid to improve the performance of key DSP components like the multiplier, largely due to its importance in the performance of the whole baseband DSP system.

This thesis presents a novel retargetable multiplication component with high throughput and power efficiency. This component was developed based on the study of (i) low-power wireless multimedia communication systems and (ii) addition and multiplication schemes for baseband DSP components. It is capable of handling both 8- and 12-bit operands, and switching between radix-4 and 8 Booth recoding schemes. Structurally, this unit incorporates various architectural level techniques including pipelining, parallelism, etc., and suitable addition schemes like carry save and carry lookahead designs. Overall, it has eight pipelining stages. Meanwhile, this unit takes the form of a soft embedded core and serves as a reusable module for System-on-a-Chip (SoC) designs. During the development process, design for reusability methodology was adopted to ensure the fulfillment of desired features. Extensive verification and simulation proved that this unit performs correct retargetable operation, achieved over 150 MIPS throughput, and obtained at least a 50% chance of reducing the number of additions needed for multiplication as compared to those of the conventional scheme.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Retargetable Arithmetic Architecture for Low-Power Baseband DSP Supporting the Design for Reusability Methodology** submitted by **Hongfan Wang** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Martin Margala

Dr. Nelson G. Durdle

Dr. Douglas M. Gingrich

Date: September 25, 2000

Acknowledgements

I would like to express my sincere gratitude and appreciation towards my supervisor, Dr. Martin Margala, for his support and guidance throughout my research process. Also, I am grateful for the productive and friendly environment created by my fellow students and colleagues in the Department and TR*Labs*. And most importantly, I would like to thank my wife for her understanding and support.

Finally, I would like to acknowledge and thank the support from TR*Labs* (Telecommunications Research Labs) and CMC (Canadian Microelectronic Corporation).

Table of Contents

CHAPTER 1 INTRODUCTION 1

1.1 Wireless Multimedia Communication Systems 2

1.2 Digital Signal Processing Systems 3

1.3 Research Objectives 4

1.4 Thesis Organization 7

CHAPTER 2 LOW-POWER WIRELESS MULTIMEDIA COMMUNICATION SYSTEMS 8

2.1 Low-Power Wireless Communication Protocol Design 9

2.1.1 Data Link Layer 10

2.1.2 Transport Layer 11

2.1.3 Formal Design Approach 12

2.1.4 Miscellaneous 12

2.2 Low-Power Front-end Design 13

2.2.1 Heterodyne Architecture 14

2.2.2 Homodyne Architecture 14

2.2.3 Low-IF Architecture 15

2.3 Low-Power Baseband Design 16

2.3.1 Algorithmic Level Low-Power DSP Design 16

2.3.2 Architectural Level Low-Power DSP Design 17

2.3.3 Lower Level Low-Power DSP Design 17

2.3.4 Conclusions 18

CHAPTER 3 ADDITION AND MULTIPLICATION SCHEMES FOR BASEBAND DIGITAL SIGNAL PROCESSING COMPONENTS 19

3.1 Addition 20

3.1.1 Basic Operations 20

3.1.1.1 Half Adder 20

3.1.1.2 Full Adder 21

3.1.2 Implementation Schemes 21

3.1.2.1 Bit Serial Adder 21

3.1.2.2 Carry Ripple Adder 22

3.1.2.3 Carry Completion Adder 23

3.1.2.4 Carry Skip Adder 24

3.1.2.5 Carry Lookahead Adder 26

3.1.2.6 Manchester Adder 28

3.1.2.7 Conditional Sum Adder 29

3.1.2.8 Carry Select Adder 30

3.1.2.9 Carry Save Adder 31

3.1.2.10 Digit-Serial Adder 33

3.1.2.11 Redundant Number Addition 34

3.2 Multiplication 35

3.2.1 Basic Operations 35

3.2.2 Implementation Schemes 35

3.2.2.1 Multiple Scan & Shift Multiplication 35

3.2.2.2 String Recoding and Booth Multiplier 36

3.2.2.3 Canonical Recoding 39

3.2.2.4 Array Multiplier and its Modification 39

3.2.2.5 Wallace Tree / Dadda Multiplier 40

3.2.2.6 Digit-Serial Multiplier 44

3.2.2.7 Other Array Multipliers 44

3.2.2.8 ROM-Adder Multiplication Networks 44

3.2.2.9 Logarithmic Multiplication 45

3.3 Circuit Design Styles 45

3.3.1 Static CMOS Logic 46

3.3.2 CMOS Transmission Gate Logic 46

3.3.3 Complementary Pass-Transistor Logic and Double Pass-Transistor Logic 46

3.3.4 Cascode Voltage Switch Logic 46

3.3.5 Differential Cascode Voltage Switch Logic 47

3.3.6 Other Circuit Design Styles 47

3.4 Characterization 47

3.5 Conclusions 48

CHAPTER 4 RETARGETABLE ARITHMETIC ARCHITECTURE FOR LOW-POWER BASEBAND DSP SUPPORTING THE DESIGN FOR REUSABILITY METHODOLOGY 49

4.1 Introduction 49

4.2 Features of the Multiplication Component 50

4.3 Structures of the Reusable Multiplication Component 51
4.3.1 Overview 51
4.3.2 Recoding Section 53
4.3.2.1 Code Generate Stage 53
4.3.2.2 Enable Generate Stage 55
4.3.2.3 Value Generate Stage 57
4.3.2.4 Partial Product Generate Stage 57
4.3.3 Custom Addition Section #1 60
4.3.3.1 First Carry Save Addition Stage 60
4.3.3.2 First Carry Propagate Addition Stage 62
4.3.4 Custom Addition Section #2 64
4.3.4.1 Second Carry Save Addition Stage 64
4.3.4.2 Second Carry Propagate Addition Stage 65
4.4 Design Methodology 67
4.4.1 Design Flow 67
4.4.2 Timing and Synthesis Considerations 69
4.5 Results 70
4.5.1 Retargetability 70
4.5.2 Throughput 71
4.5.2.1 Timing Report for Code Generate Stage 72
4.5.2.2 Timing Report for Enable Generate Stage 73
4.5.2.3 Timing Report for Value Generate Stage 74
4.5.2.4 Timing Report for Partial Product Generate Stage 76

4.5.2.5 Timing Report for First Carry Save Addition Stage 77

4.5.2.6 Timing Report for First Carry Propagate Addition Stage 78

4.5.2.7 Timing Report for Second Carry Save Addition Stage 80

4.5.2.8 Timing Report for Second Carry Propagate Addition Stage 81

4.5.2.9 Summary of Maximum Delay for Each Pipeline Stage 83

4.5.3 Power 84

4.6 Design for Testability of the Multiplication Component 85

4.6.1 Scannable Registers 85

4.6.2 AND/OR Tree Based Technique 86

4.6.3 C-testable Technique for Iterative Logic Arrays 87

4.6.4 Application to the Multiplication Component 88

4.6.4.1 AND/OR Tree for the Recoding Circuit 88

4.6.4.2 C-testable Technique for the Adders 89

CHAPTER 5 CONCLUSIONS AND RECOMMENDATIONS 90

BIBLIOGRAPHY 93

APPENDICES 102

List of Tables

Table 1.1: Overview of characteristics of the multiplication component. 5

Table 3.1: Radix-4 Booth recoding algorithm. 37

Table 3.2: Radix-8 Booth recoding algorithm. 37

Table 3.3: Radix-4 canonical multiplier recoding algorithm. 39

Table 4.1: Function of the control signal. 53

Table 4.2: Correspondence between cod_i and the partial product needed. 55

Table 4.3: Maximum delay of each pipeline stage. 83

Table 4.4 Simulation result comparing addition needed with and without recoding. 84

List of Figures

Figure 1.1: RF design hexagon. 3

Figure 1.2: Schematic of a digital signal processing system. 4

Figure 2.1: A hybrid reference model. 10

Figure 3.1: Schematic of a bit serial adder. 22

Figure 3.2: Schematic of an 8-bit carry ripple adder. 23

Figure 3.3: Schematic of a carry completion adder. 24

Figure 3.4: Schematic of a carry propagation (CP) cell. 24

Figure 3.5: 16-bit constant-block-width carry skip adder. 25

Figure 3.6: 16-bit variable-block-width carry skip adder. 25

Figure 3.7: Schematic of a 4-bit carry lookahead adder. 27

Figure 3.8: A 16-bit adder using hierarchical carry lookahead. 28

Figure 3.9: Schematic of a 4-bit Manchester adder. 29

Figure 3.10: Schematic of a 4-bit conflict free Manchester adder. 29

Figure 3.11: Schematic of an 8-bit conditional sum adder. 30

Figure 3.12: Schematic of an 8-bit carry select adder. 31

Figure 3.13: Schematic of a carry save adder. 32

Figure 3.14: Schematic of a pipelined carry propagate adder (CPA). 33

Figure 3.15: Carry save and propagate adder using 2-bit multiplier scan. 35

Figure 3.16: Schematic of a 4-bit scanning Booth multiplier using CSA/CPA. 38

Figure 3.17: Schematic of an 8-bit array multiplier, with a unit cell. 41

Figure 3.18: Schematic of an 8-bit Wallace tree multiplier. 42

Figure 3.19: Reduction scheme of an 8-bit Dadda multiplier. 43

Figure 4.1: Illustrative diagram of main structures of the multiplication component. 52

Figure 4.2: Code generate stage. 54

Figure 4.3: Enable generate stage. 56

Figure 4.4: Value generate stage. 58

Figure 4.5: Partial product generate stage. 59

Figure 4.6: First carry save addition stage. 61

Figure 4.7: First carry propagate addition stage. 63

Figure 4.8: Second carry save addition stage. 65

Figure 4.9: Second carry propagate addition stage. 66

Figure 4.10: Traditional ASIC design flow. 67

Figure 4.11: Core design flow. 68

Figure 4.12: Timing report for code generate stage. 73

Figure 4.13: Timing report for enable generate stage. 74

Figure 4.14: Timing report for value generate stage. 76

Figure 4.15: Timing report for partial product generate stage. 77

Figure 4.16: Timing report for first carry save addition stage. 78

Figure 4.17: Timing report for first carry propagate addition stage. 80

Figure 4.18: Timing report for second carry save addition stage. 81

Figure 4.19: Timing report for second carry propagate addition stage. 83

Figure 4.20: Scannable registers. 86

Figure 4.21: ADD/OR tree based structure. 87

Figure 4.22: C-testable structure for an ILA. 87

Figure 4.23: Coding circuit with AND/OR trees. 88

Figure 4.24: Application of C-testable technique to a CLA adder. 89

Chapter 1

Introduction

Given the rapid development of wireless multimedia communication devices, various digital signal processing (DSP) systems are under intensive investigation for better adaptability, higher throughput, and less power consumption [1]. These demands, in turn, are prompting research on algorithmic and architectural levels during DSP system design. From a hardware perspective, various DSP components are rigorously studied in order to achieve better overall system performance. Among them, addition and multiplication components are major parts in all DSP systems and are considered bottleneck of improvements in the aforementioned areas.

This thesis research was to develop a novel retargetable, high throughput and power-efficient multiplication component for possible use in DSP systems. Besides having these

functional features, it was also developed through the adoption of a formal design methodology and takes the form of a reusable module ready for incorporation into various digital systems. As will be addressed in a later chapter, this approach is a promising way of tackling the ever-increasing gate-count and system complexity problems facing contemporary hardware designers.

1.1 Wireless Multimedia Communication Systems

Generally speaking, wireless multimedia communication systems include the adopted wireless communication protocols, along with wireless terminals and base stations that physically transmit and receive information between communicating parties. Hardware designers normally pay attention to wireless terminals, which can be divided into front-end and baseband portions although exact definition varies, and the quantification of their interactions is still under investigation [2]. The front-end portion handles radio frequency signals and normally includes low noise amplifier, mixer and image rejection filter, etc., while the baseband portion deals with low-frequency analog and digital signal processing tasks.

Due to the many design tradeoffs [3] (Figure 1.1 [4]), as well as designer expertise requirements and the lack of suitable CAD tools, the front-end portion of the wireless terminal is still the design bottleneck of the entire system although the baseband portion involves far more components.

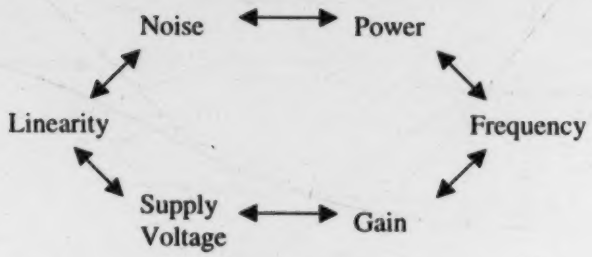


Figure 1.1 RF design hexagon.

In the baseband portion of a wireless terminal, mainly DSP systems carry out various functions so as to facilitate wireless multimedia communications. These functions can be categorized into (i) communication functions like speech coding, channel coding, demodulation, equalization, etc.; and (ii) multimedia user interface functions like audio compression, video compression, speech and handwriting recognition, and speech synthesis. (Some of the functions are not yet available on current commercial wireless terminals, but are certain to appear in the future.) It is also expected that more signal processing tasks will be shifted from the front-end to the baseband in order to make up for the inherent limitations in the front-end.

1.2 Digital Signal Processing Systems

Digital signal processing systems (Figure 1.2) generally include an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC), which connect the digital signal processor with real-world analog signals. Within the digital signal processor are memory elements and addition and multiplication components [5].

Figure 1.2 Schematic of a digital signal processing system.

Digital signal processing systems have rapidly developed in the past several decades, largely due to their inherent advantages over analog signal processing systems and the advances in very-large-scale-integration (VLSI) technology. These advantages include much better (i) immunity towards environmental and aging effects, (ii) flexibility in reconfiguration and adaptation for applications, and (iii) control over accuracy, etc. On the other hand, the drawbacks of DSP systems compared to analog signal processing systems are (i) the need for ADC and DAC, and (ii) the possibility that throughput requirements for digital components may exceed those available from current state-of-the-art VLSI technology. This thesis research partially addressed the latter drawback.

1.3 Research Objectives

This thesis research focused on the research and development of a novel multiplication component for possible use in baseband DSP systems. The uniqueness of this component is demonstrated in three areas: (i) functionality, (ii) structure, and (iii) design methodology, as summarized in the following table.

Functionality	• Retargetability Expected to be capable of handling both 8- and 12-bit operands and switching between radix-4 and 8 Booth recoding schemes • High Throughput Expected throughput exceeding 150 MIPS • Low Power Operation Expected power consumption less than that required for conventional multiplier, i.e., array multiplier
Structure	• Architecture Incorporation of various architectural level techniques, i.e., pipelining, parallelism, etc. • Addition Components Selective usage of suitable addition schemes within certain pipelining stages, i.e., carry save addition, carry lookahead addition, etc.
Design Methodology	• Design Flow Top-down, yet recursive approach • Timing, Synthesis and DFT Considerations Robust and reliable core design

Table 1.1 Overview of characteristics of the multiplication component.

Functionality: Retargetability is provided to avoid multiple multiplier inclusion in cases when different wordlengths and Booth recoding schemes are required, thus increasing overall system efficiency. For example, when both radix-4 and 8 Booth recoding schemes are used from time to time, this multiplication component can be adopted instead of using two separate multipliers. Thus, substantial savings in hardware can be achieved resulting from hardware sharing within this component. On the other hand, since this component is to be incorporated into baseband DSP systems, high throughput, which in this project is over 150 MIPS, is desired to handle the increasing signal processing tasks needed within the mobile terminal. This retargetability and throughput requirement should also be obtained at the expense of less power consumption as compared to that of conventional multipliers, because the increase in battery power occurs at a much slower pace.

Structure: The structure of this multiplication component was developed in order to meet the above-mentioned functionality requirements, after a comprehensive literature review and comparison between different implementation schemes. From an architectural point of view, it involved the combination of various architectural level techniques, like pipelining and parallelism. Also, within certain pipeline stages, a combination of different addition schemes were used, including carry save addition and carry lookahead addition.

Design Methodology: This multiplication component was developed in the form of a reusable soft embedded core following the design methodology, which included an enhanced design flow, as well as timing, synthesis and DFT considerations. This design methodology was adopted to produce a developed component for possible incorporation into System-on-a-Chip (SoC) designs. The SoC design approach, which will be further

described in Section 4.1, provides an efficient way for designing system with no inalienable discrete components. It is gaining increasing popularity in the semiconductor industry.

1.4 Thesis Organization

This thesis is organized into five chapters. Chapter 1 provides a brief explanation of the motivation for undertaking a research project about multiplication components for baseband DSP systems. Chapters 2 and 3 explain the background information in more detail and result from comprehensive literature reviews on broader topics. Among them, low-power wireless multimedia communication systems are discussed in Chapter 2 and addition and multiplication schemes for baseband DSP components are reviewed in Chapter 3. Chapter 4 describes the research and development of a novel multiplication component for baseband DSP systems. It includes (i) introduction to SoC designs, (ii) features and (iii) structures of the reusable multiplication component, (iv) design methodology, (v) results, and (vi) design-for-testability (DFT) considerations. Finally, Chapter 5 provides conclusions for this thesis research and makes recommendations for research in the future.

Chapter 2

Low-Power Wireless Multimedia Communication Systems*

Currently, much of the information technology (IT) industry's attention is being paid to low-power wireless multimedia communication systems, largely due to the surging demand for better connectivity and stronger mobile computing ability, and the relatively slower progression in battery power [6]. Design approaches for low-power wireless multimedia communication systems normally fall into three areas: (i) wireless communication protocols used by mobile terminals and the wireless communication

* This section is part of a published paper. Hongfan Wang and Martin Margala, *Low-Power Wireless Multimedia Communication Systems*, Proceedings of the 2000 Canadian Conference on Electrical and Computer Engineering, May 2000, pp. 1063 – 1067.

network, (ii) the front-end portion, and (iii) the baseband portion of the mobile terminal. (However, the quantification of interactions between these two portions is still under investigation.) These design approaches are reviewed in this chapter.

In the communication protocols area, low-power design approaches for data link protocols, especially the medium access control (MAC) protocol, and for network protocols are addressed, along with formal design suggestions for communication protocols and the practice of software partitioning.

In the mobile terminal front-end area, various architectures are reviewed, including heterodyne, homodyne, and low-IF (intermediate frequency), along with their implications for integration and power consumption.

In the mobile terminal baseband area, various signal processing units carry out communication functions and multimedia user interface functions. General approaches for low-power digital signal processing system design are addressed.

2.1 Low-Power Wireless Communication Protocol Design

The research on designing low-power communication systems used to be in the hardware domain, focusing on various components in a mobile terminal, like the transmitter, receiver and the baseband signal processing units. However, significant power saving can also be achieved through tailoring the protocols used by wireless communication networks according to the environment they operate in. Basically, most networks are organized in a layered structure, with each layer having a specific function, while protocols are designed as rules and conventions for effective and robust electronic

communication. An easy to understand (though neither OSI nor TCP/IP) hybrid reference model is shown in Figure 2.1 [7]. Currently, more attention is being paid to the data link layer and transport layer in which fine-tuning protocols can achieve more power-saving.

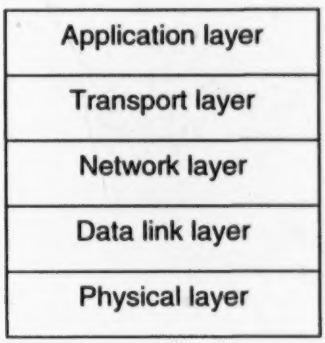


Figure 2.1 A hybrid reference model.

2.1.1 Data link layer

Generally speaking, the data link layer provides a well-defined service interface to the network layer [7]. For wireless communications, this process involves grouping network layer packets into frames, wireless link error control, and wireless flow control to facilitate the data transportation between the network layers of the two communicating parties.

In terms of low-power design research, the data link layer is receiving more interest than the transport layer, while within the data link layer, the power-conserving MAC protocols are most sought after.

The MAC protocol allocates the multi-access wireless channel among competing mobile users. Some power-conserving design guidelines for the MAC protocol were suggested [8, 9], including (i) eliminating collisions and the resulting retransmission; (ii) broadcasting the data transmission schedule for mobiles, so that they can switch to the standby mode according to the schedule; (iii) buffering packets by the base station until the mobile becomes active and decides to receive them; (iv) allocating contiguous slots for the mobile to transmit and receive data so as to minimize the switching between these two modes; and (v) using a single packet to reserve bandwidth in multiple cells. These suggestions are based on the fact that the transmission mode consumes much more power than the reception mode, while little power is spent during the standby mode. Also, much power is spent during the switching between the transmission and reception modes.

Some power-conserving MAC protocols were proposed, and a comparison between some existing MAC protocols was carried out [9] and showed that less contention normally leads to lower power consumption. These MAC protocols include IEEE 802.11 [10], PRMA (Packet reservation multiple access) [11], MDR-TDMA (Multiservices dynamic reservation TDMA) [12], EC-MAC (Energy-conserving MAC) [13], and DQRUMA (Distributed-queuing request update multiple access) [14].

2.1.2 Transport layer

The transport layer provides reliable, cost-effective data transportation between the communicating parties [7], a function achieved through the adoption of transport protocols like TCP (Transmission Control Protocol). They handle error control, flow

control, etc., with the former being extremely important for wireless communications due to the inherent high error rate resulting from various noise sources and signal fading.

Some power-conserving design guidelines for transport protocols were suggested [8], including (i) incorporating effective error-control mechanisms, and (ii) splitting the transport connection into wired and wireless network portions. Both of these guidelines aim at reducing retransmission.

2.1.3 Formal design approach

In order to facilitate formal verification and architectural-level exploration, a formal design approach for wireless communication protocols was suggested [15]. The process includes (i) formal specification, (ii) detailed extended finite state machine (EFSM) model development, (iii) debugging through high-level simulation and formal verification, (iv) formal verification and performance estimation, and (v) implementation.

2.1.4 Miscellaneous

Besides the above-mentioned protocols, other techniques like software partitioning can be adopted in the application layer for building low-power wireless multimedia communication systems. In a software partitioned system, the base station performs power-intensive computation, while the mobile supports the operation of the wireless link, and the acquisition and display of multimedia data [16].

Several prototypes have been made. (i) ParcTab [17], serving as a preliminary testbed for ubiquitous computing, integrates a palm-sized mobile computer into an office network. During the design, attention was paid to reduce the cost and size of the mobile (tab), and most of the general-purpose computing is performed on servers. (ii) InfoPad [16], an indoor mobile multimedia system, emphasizes the low power operation of the mobile (pad), and all general purpose computing is performed on servers.

2.2 Low-Power Front-End Design

Besides tailoring communication protocols for power conservation, researchers still focus much attention upon wireless terminals, trying to reduce their power consumption. As mentioned in Chapter 1, the front-end portion of the wireless terminal handles radio frequency signals and normally includes low noise amplifier, mixer and image rejection filter, while the baseband portion is concerned with low-frequency analog and digital signal processing.

For low-power front-end design, higher integration and miniaturization are always desirable [18], although current technology normally requires discrete or bulky parts (e.g., inductors) for bandpass amplifiers and filters in order to meet performance specifications [19]. This review, however, focuses on architectural level exploration, since it can lead to possible removal of these discrete or bulky components, which generally consume much power and yield orders of magnitude in power reduction, as demonstrated by the development of radio [20, 21] and paging [22] receivers.

Most commonly used architectures are briefly addressed, which include heterodyne, homodyne, and low-IF architectures, with their limitations and implications for integration and miniaturization stressed.

2.2.1 Heterodyne architecture

In heterodyne architectures, the desired signal in a high-frequency narrow channel is downconverted from its carrier frequency to IF through a mixer (converter) to make it feasible to filter out strong interferers surrounding the signal channel [83]. After the first downconversion, the signal can be demodulated or further downconverted.

Heterodyne architectures are the most commonly used transceiver front-end architecture. Their major problem is that of mirror frequency resulting from the mixer's inability to differentiate the polarity of frequency differences. Consequently the transceiver needs at least one high-quality, tunablem high-frequency filter, which is still available only in discrete form. Also, high-frequency amplifiers are normally required and are not very power-efficient.

2.2.2 Homodyne architecture

In homodyne architectures, the desired signal is directly downconverted from its carrier frequency to the baseband through simple or quadrature downconversion with local oscillator (LO) frequency equal to the carrier frequency, and the IF becomes zero [83]. Quadrature downconversion generates both in-phase (I) and quadrature (Q) components

of the signal while translating the spectrum to zero frequency, and works for frequency- and phase-modulated signals. Simple downconversion can be used for double-sideband AM signals.

Homodyne architectures have no mirror frequency problem, and thus a low-quality, broadband, high-frequency filter can be used. After downconversion, lowpass filters and baseband amplifiers can be used and are ready for integration. However, a serious problem associated with this architecture is DC offset caused by transistor mismatch, LO leakage to input, or rectification due to even order distortion. Besides problems to itself, radiation due to leakage of the LO signal to the antenna can cause interference to other receivers in the same band and using the same wireless standard [4].

2.2.3 Low-IF architecture

In low-IF architectures, the same downconversion principle is used, except that the IF is chosen to be in the range of hundreds of kHz to a few MHz, instead of zero Hz [23][83].

This architecture can also have a high degree of integration, since filtering is mainly carried out at low frequency. Moreover, DC offset can be avoided if the downconverted signal channel does not cover zero frequency. However, in-band image rejection requires high-performance image reject mixers and is still under investigation [19].

Due to the inherent tradeoffs in key front-end components and the quick sophistication of digital signal processing units in the baseband, it is expected that more signal processing tasks will be shifted to the baseband so as to make up for the imperfections in the front-end.

2.3 Low-Power Baseband Design

In the baseband portion of a wireless terminal, most communication functions, like speech coding, channel coding, demodulation, and equalization, and multimedia user interface functions like audio compression, video compression, speech and handwriting recognition, and speech synthesis, (some are to appear in the future), are carried out through various digital signal processing systems.

For the predominant CMOS digital circuits, power consumption comes from mainly three sources, as shown in the following equation [24, 25]:

$$P_{total} = P_t \bullet (C_L \bullet V \bullet V_{dd} \bullet f_{clk}) + I_{sc} \bullet V_{dd} + I_{leakage} \bullet V_{dd} \qquad (2.1)$$

where P_{total} is the total power dissipation, P_t is the activity factor, C_L is the loading capacitance, V is the voltage swing, V_{dd} is the supply voltage, f_{clk} is the clock frequency, I_{sc} is the direct-path short circuit current, and $I_{leakage}$ is the substrate injection current and subthreshold current. Normally, the first component of the equation plays a major role in total power dissipation; thus making algorithmic and architectural level exploration highly desirable and effective for low-power digital design. Much attention was spent on this area while designing low-power DSP systems.

2.3.1 Algorithmic level low-power DSP design

Algorithmic level low-power DSP design techniques include pipelining [24, 26], retiming [27], unfolding [28, 29], loop-unrolling [30], look-ahead, [31], and algebraic

transformation [32], and comprehensive reviews have been published [33, 34]. Most of these techniques unveil or create concurrency to increase throughput and allow voltage scaling, but these are achieved at the expense of larger silicon area consumption. For example, pipelining involves delay element insertion at appropriate intermediate points in the data-flow graph of an algorithm/structure to facilitate concurrent signal processing. Retiming involves moving around delays in a data-flow graph while not changing the computation and can be used to reduce the critical path of the data-flow graph.

From another perspective, algorithmic level low power design can be pursued through: (i) reducing the switched capacitance by minimizing the complexity of the system; (ii) reducing switching activity by data coding [35].

2.3.2 Architectural level low-power DSP design

Architectural level low-power DSP design techniques include pipelining [24, 26], parallelism, [24], distributed processing [35, 36], and dynamic and static power management [35]. From a hardware design perceptive, these techniques lie at the second highest level of abstraction, the "register-transfer level" (RTL), in which the system is described in terms of data storage and transformation units.

2.3.3 Lower level low-power DSP design

Besides the investigation of the algorithmic and architectural levels, low power DSP design can also benefit from lower level techniques such as technology optimization, as well as physical, circuit and logic style optimization [24]. Detailed discussions about

these fields can be found in the references [35, 37, 38]. However, these optimizations often reduce the power consumption of DSP systems in a rather indirect way by improving the power performance of various DSP sub-components.

2.4 Conclusions

Major issues in the design of low-power wireless multimedia communication systems have been briefly reviewed. Generally speaking, wireless communication protocol selection and design should reflect power-conserving principles, e.g., minimizing retransmission, shortening active mode duration, and decreasing mode switching, to preserve battery power on the mobile terminal. Meanwhile, the front-end of mobile terminals must strive for a higher level of integration and miniaturization, possibly through architectural innovation, so as to reduce power consumption while still meeting prescribed specifications and maintaining a certain quality of service (QoS). On the other hand, power reduction in the baseband of mobile terminals can take advantage of the developments in low-power CMOS digital design, while higher level algorithmic and/or architectural modifications provide more options for much better power efficiency.

Chapter 3

Addition and Multiplication Schemes for Baseband Digital Signal Processing Components*

Attention normally focuses on three major areas of contemporary VLSI designs: performance, area, and energy consumption. Different applications have different criteria in terms of performance. For general-purpose processors, achieving maximum computation speed is always desirable, while for various baseband digital signal

* A version of this section has been published. Hongfan Wang and Martin Margala, *Addition and Multiplication Scheme for Energy-Efficient DSP Component*, Proceedings of the 2000 Canadian Conference on Electrical and Computer Engineering, May 2000, pp. 636 – 641.

processing (DSP) applications, catering for their throughput requirements is a more realistic choice. Besides performance, minimization of the die area is also a much sought-after attribute. In recent years, power consumption has become a major concern for many applications. Obtaining minimal power consumption is not only attractive to elongate battery life in wireless devices, but also useful to reduce the cost associated with cooling for conventional desktop devices.

In this chapter, the design of various fixed-number addition and multiplication systems is reviewed. Different addition and multiplication schemes are presented along with their implication for performance, area, and power consumption. Also briefly mentioned are circuit design styles, characterization criteria and test strategies.

3.1 Addition

3.1.1 Basic Operations

In the following two sections, basic addition operation will be illustrated using half adder and full adder models.

3.1.1.1 Half Adder

The half adder is the simplest and most primitive arithmetic system. It gets two input bits and produces a sum and a carry-out bit. The relationship between the outputs and inputs is represented in Equations (3.1) and (3.2).

$$Sum = A \oplus B \tag{3.1}$$

$$Carry = A \bullet B \tag{3.2}$$

3.1.1.2 Full Adder

For multi-bit addition, half adders are not enough, since the carry-in from a proceeding bit addition should also be taken into account. A full adder operation can thus be defined as in Equations (3.3) and (3.4).

$$S_i = A_i \oplus B_i \oplus C_{i-1} \tag{3.3}$$

$$C_{i+1} = A_i \bullet B_i + C_i \bullet A_i + C_i \bullet B_i = A_i \bullet B_i + C_i \bullet (A_i + B_i) \tag{3.4}$$

In order to achieve desirable performance, area, and energy consumption, different implementation schemes have been proposed and adopted.

3.1.2 Implementation Schemes

3.1.2.1 Bit Serial Adder

In a bit serial adder, operands are taken in bit-by-bit with results produced in the same manner. Memory elements are used to hold intermediate and final results (Figure 3.1). Three registers, A, B and S, are all shift registers, among which, A and B hold two operands, while S holds the sum. A delay element, normally implemented using a D-flipflop, is used to ensure proper alignment of carry-out with adjacent input bit pairs.

The performance of a bit serial adder is low. However, it consumes the least area among its peers and is considered when area is of utmost importance. Another way of using it is to operate many concurrently, as in some parallel SIMD machines [40].

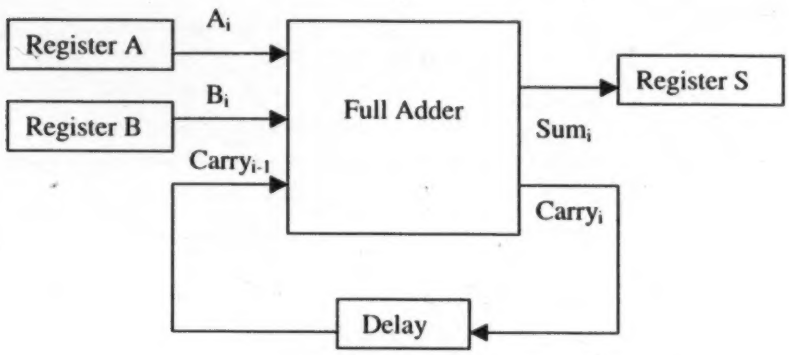


Figure 3.1 Schematic of a bit serial adder.

In order to achieve a better performance, various kinds of bit parallel adders were proposed.

3.1.2.2 Carry Ripple Adder

Among all adders, the carry ripple adder [26] most resembles a human being carrying out addition by using paper and pencil. An n-bit carry ripple adder consists of n full adders with their carries connected as in a linked list fashion (Figure 3.2 [41]).

A carry ripple adder does not yield a much better performance than that of a bit serial adder, despite a much heavier investment in hardware, largely due to the long path of carry propagation.

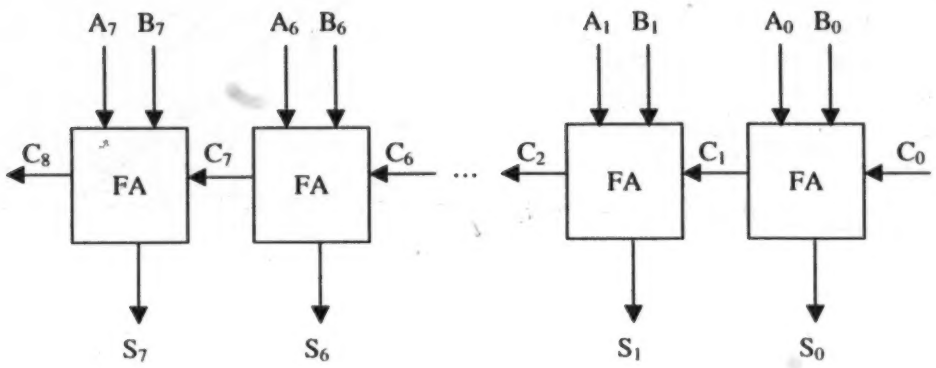


Figure 3.2 Schematic of an 8-bit carry ripple adder.

3.1.2.3 Carry Completion Adder

The carry completion adder [42, 54] can be constructed from a carry ripple adder through the incorporation of carry-propagation-complete detection logic (Figures 3.3 and 3.4). The underlying idea is to determine the length of the critical path, according to the worst case carry propagation required for each pair of operands. When the carry completion signal is asserted, the addition process will be complete after a delay equivalent to the latency of a full adder.

However, there are two practical concerns: the high fan-in required of the carry completion AND gate, and the asynchronous operation of the adder, which may complicate the overall synchronous system design, so that the resynchronization time may well outweigh any gain from using this design approach.

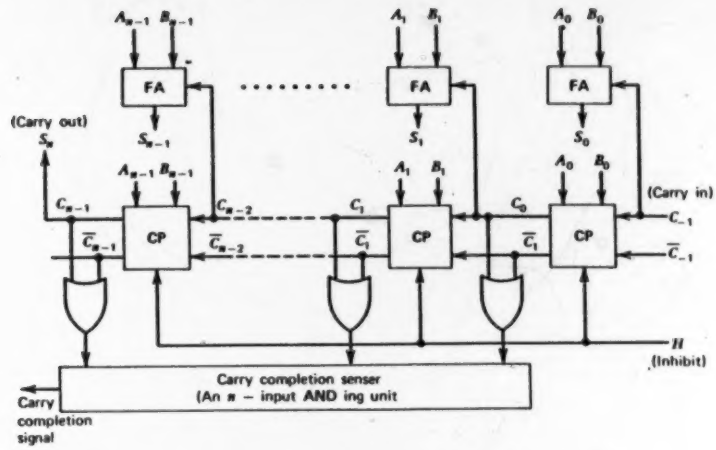


Figure 3.3 Schematic of a carry completion adder.

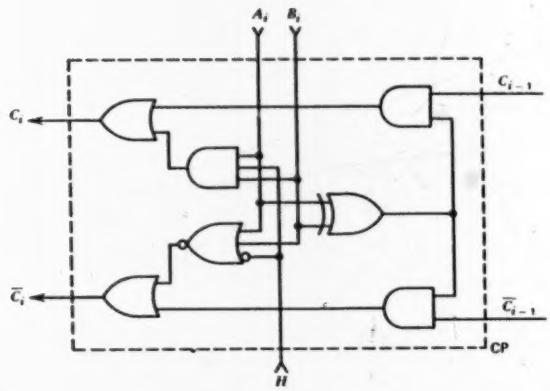


Figure 3.4 Schematic of a carry propagation (CP) cell.

3.1.2.4 Carry Skip Adder

The principal idea for carry skip adders is that if corresponding bits in the two operands are not equal, carry-in passed into that bit position will propagate to the next bit position. The adder is designed by making a carry-in skip any block for which it is known that the carry will propagate through.

The carry skip adder has two variations, shown in Figures 3.5 and 3.6 [41], respectively: (i) the constant-block-width carry skip adder [26], which uses only a single level of skip logic, where the optimal width of each block is determined by a formula based on the number of bits to be added; and (ii) the variable-block-width carry skip adder [43], in which multiple levels of skip logic and variable block sizes are used. Both schemes make it possible to determine the carryout from each block before the calculation inside each block is done, thus resulting in faster addition.

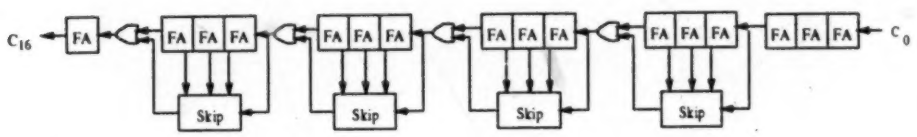


Figure 3.5 16-bit constant-block-width carry skip adder.

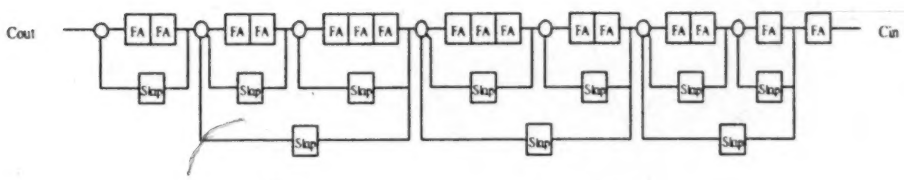


Figure 3.6 16-bit variable-block-width carry skip adder.

Although the theoretical performance of carry skip adders is not among the highest, they can be combined with other schemes, i.e., Manchester adders, to achieve adder structures of high regularity, (only two kinds of modules are needed, i.e., FA and Skip); thus, carry skip adders can be made desirable for VLSI implementation.

3.1.2.5 Carry Lookahead Adder (CLA)

Carry lookahead is a parallel carry generation scheme [44, 45] for speeding up addition. Addition at each operand bit will produce a carry because either it is generated at this bit position, or one is propagating from the preceding bit position. A carry is generated if both operand bits are 1, and it is propagated if one of the operand bits is 1 and the other is 0. This observation is shown in the definition of the following two auxiliary functions:

$$G_i = A_i \bullet B_i \tag{3.7}$$

$$P_i = A_i \oplus B_i \tag{3.8}$$

The carry-generate function G_i reflects the condition that a carry is originated at the ith bit position. The carry-propagate function P_I is asserted when the ith bit position will pass the incoming carry C_{i-1} to the next higher bit position. Thus, the calculation of sum and carry can be modified as

$$S_i = P_i \oplus C_{i-1} \tag{3.9}$$

$$C_i = G_i + P_i \bullet C_{i-1} \tag{3.10}$$

Practical usage of this pure CLA algorithm is limited to the smallest adders ($\leq$ 4 bits), due to the rapid increase in fan-out and fan-in requirements as the adder size grows. Modified approaches were proposed, i.e., Ripple-block CLAs, Block CLAs, etc. For ripple-block CLAs, bit-stages of the adder are grouped into blocks. The carry lookahead scheme is implemented within each block, with carries rippling from block to block. For block CLAs, the situation is the opposite, with carries rippling within each block, and lookahead being used between blocks. However, neither of these two schemes is popular for practical large adders. Instead, a scheme using hierarchical carry lookahead is adopted due to its high efficiency in terms of performance gain over hardware investment.

For a 4-bit carry lookahead block (Figure 3.7 [25]), two additional terminal functions, namely, block carry generate G^* and block carry propagate P^*, are defined as

$$P^* = P_0P_1P_2P_3 \tag{3.11}$$

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 \tag{3.12}$$

($P_i \equiv$ P<i>; $G_i \equiv$ G<i>; $C_i \equiv$ C<i>). The carry out of the block can be derived as

$$C_3 = G^* + P^* \bullet CI \tag{3.13}$$

where CI is the carry into the block. (For 4-bit adders, faster carry generation can be achieved through the use of a single C_3 gate using a Manchester adder covered later.)

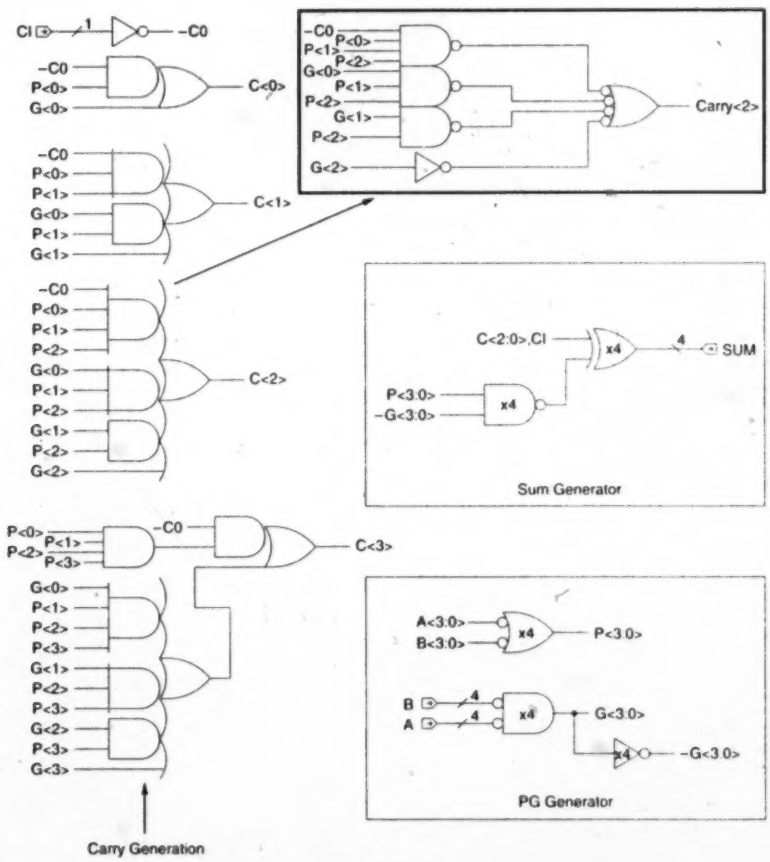


Figure 3.7 Schematic of a 4-bit carry lookahead adder.

A 16-bit adder using a hierarchical carry lookahead can then be constructed, as shown in Figure 3.8 [39], in which C_0 is the possible carryin into the whole adder, and C_0 is needed in both the 4-bit adder unit and the lookahead carry unit.

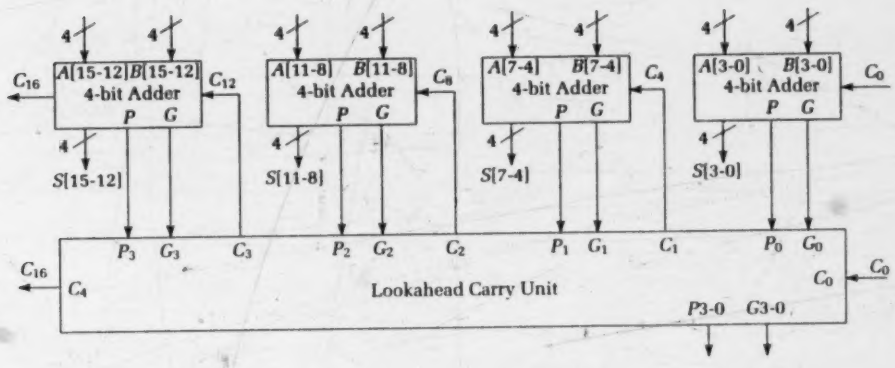


Figure 3.8 A 16-bit adder using hierarchical carry lookahead.

3.1.2.6 Manchester Adder

Basically, the Manchester adder is a modification of the carry lookahead scheme to further speed up carry generation through the use of a single carry-out gate that is implemented with multiplexers.

Using readily available generate and propagate signals, carries are produced using the iterative formula, in which P_i' is the complementary of P_i,

$$C_{i+1} = P_iC_i + G_i = P_iC_i + P_i'G_i \tag{3.14}$$

leading to an implementation based on multiplexer, shown in Figure 3.9 [25].

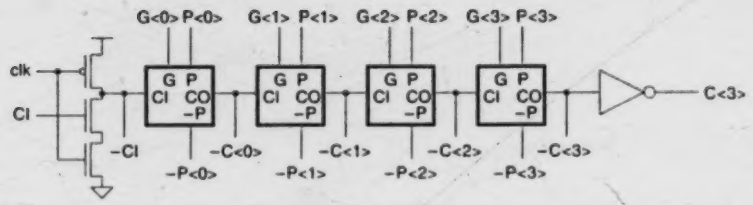


Figure 3.9 Schematic of a 4-bit Manchester adder.

A 4-bit conflict free Manchester adder implementation is shown in Figure 3.10 [25], where the control signals T_1, T_2, and T_3 are defined as

$$T_1 = -(P_0P_1P_2)P_3;\ T_2 = -P_3;\ T_3 = P_0P_1P_2P_3 \tag{3.15}$$

Very wide, fast adders may be constructed by extending the carry bypass.

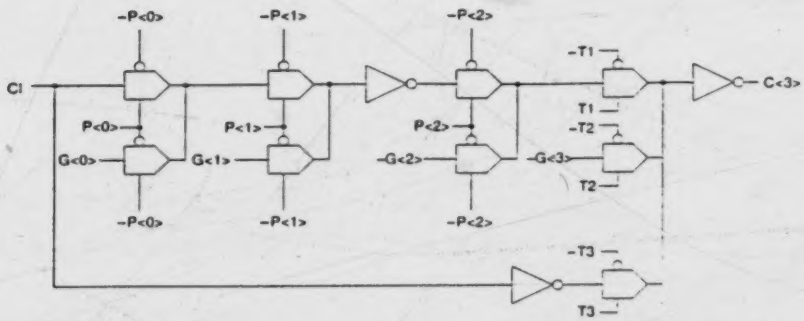


Figure 3.10 Schematic of a 4-bit conflict free Manchester adder.

3.1.2.7 Conditional Sum Adder

The general idea of a conditional sum adder is to use multiplexers to recursively combine successively larger blocks of conditional sum and carry bits (Figure 3.11 [41]). The basic

cell accepts two bits to be added and produces both sets of sum and carry for assumed carry-ins of zero and one (based on the Sklansky "H" cell [46]).

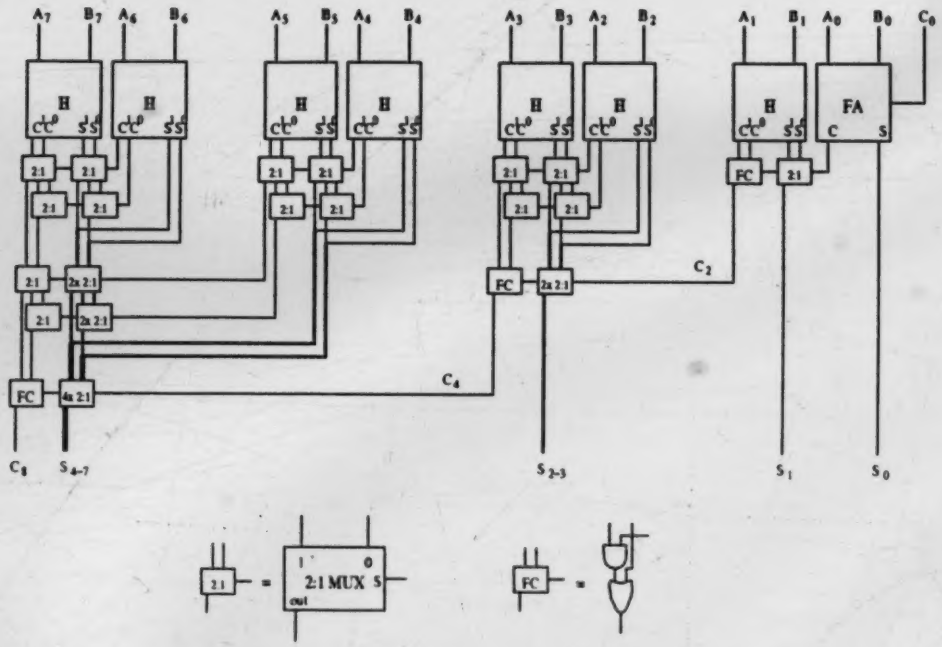


Figure 3.11 Schematic of an 8-bit conditional sum adder.

Advantages of this design include high performance, low fan-in, and the likelihood of pipelining the design. However, the large fan-out poses a problem at the last level of the adder.

3.1.2.8 Carry Select Adder

The carry select adder [47] uses the same principle as the conditional sum adder. It uses the generated distant carries to select the correct sum outputs from two simultaneously

generated provisional sums under zero and one carry input conditions. However, instead of generating bit-wise provisional sums and carriers, the carry select adder partitions a long adder into fixed-size adder sections and uses the correct carry input to select the true sum output from two sets of simultaneously produced section addition results. A simple schematic of an 8-bit carry select adder is shown in Figure 3.12 [39].

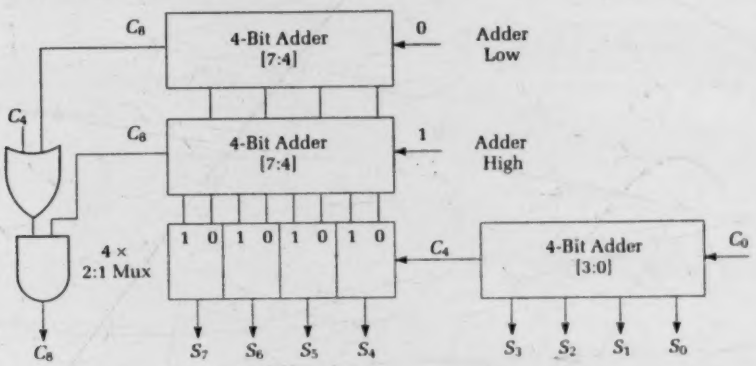


Figure 3.12 Schematic of an 8-bit carry select adder.

Generally speaking, this kind of adder provides a good example of achieving high performance at the expense of heavier hardware investment.

3.1.2.9 Carry Save Adder (CSA)

The adders described above are designed for two-operand addition. Carry save adders are used in fast addition of many binary numbers (a direct application is to add partial products during multiplication) with limited hardware. Carry propagation is avoided until all additions are completed and then takes one (or several) final cycles to complete carry propagation for all additions [48].

Practically, for an n-bit CSA, carries and sums are registered in 2n registers. Figure 3.13 [25] shows an implementation of an adder circuit, which uses two 4-bit CSAs.

Usually, a fast architecture is used for the final carry propagate adder (CPA) for computation of the final result of the sums and carries of each CSA. One scheme (Figure 3.14 [25]) is to use cascaded CSAs. Registers are used at the input and output of the CPA to ensure synchronous operation.

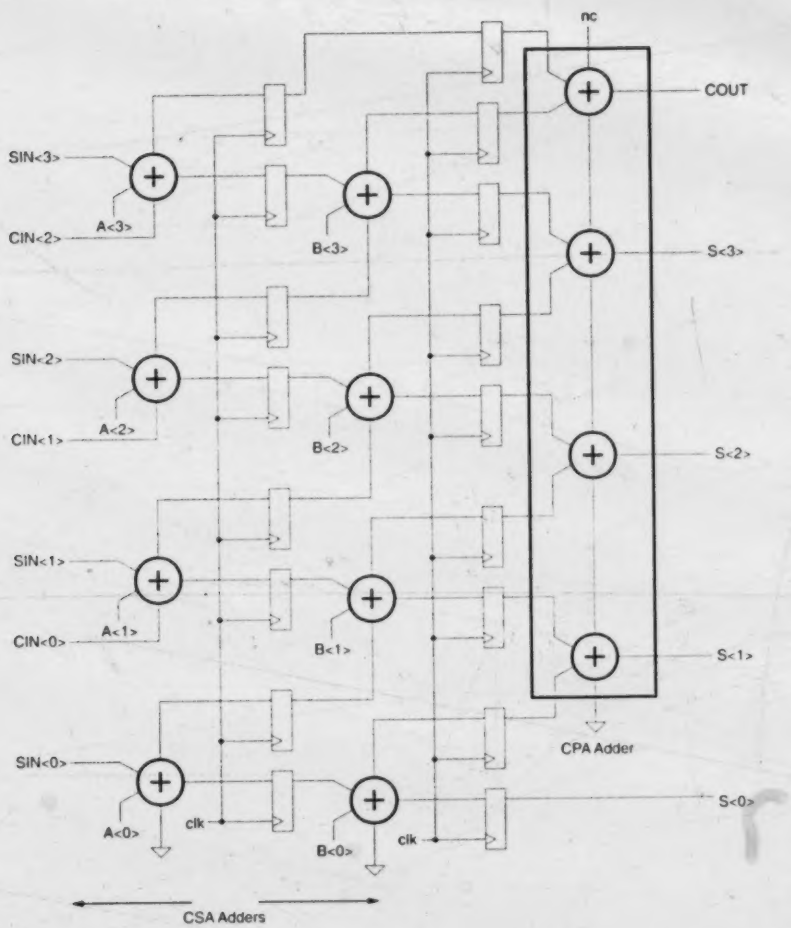


Figure 3.13 Schematic of a carry save adder.

The carry save technique is mainly used for pipelined parallel arithmetic, and its success lies mostly in DSP applications such as filtering, where achieving maximum throughput is the goal. Also, carry-save arithmetic is the basis of the well-known Wallace-tree multiplier.

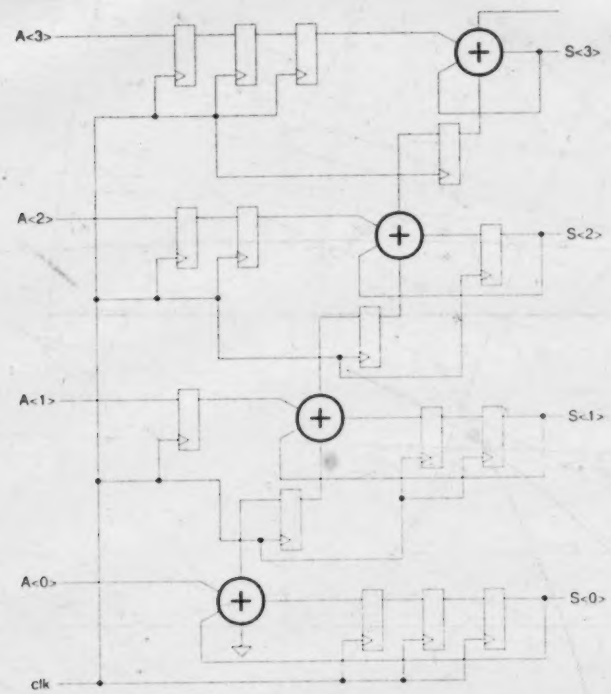


Figure 3.14 Schematic of a pipelined carry propagate adder (CPA).

3.1.2.10 Digit-Serial Adder

Generally speaking, digit-serial [49] implementation is used mainly for signal processing applications. It is suggested as an alternative choice for cases where the sampling rate

demand is too high for bit-serial systems, while bit-parallel systems require excessive hardware. The number of bits processed in a clock cycle is referred to as the "digit-size", and it is chosen to match the clock cycle of the circuit and sample rate of the application, when the design is used for real-time signal processing applications.

3.1.2.11 Redundant Number Addition

Redundant number representation allows serial operations to proceed in a most-significant-digit-first (MSD-first) mode and is mainly used in MSD-first serial arithmetic (also called on-line arithmetic). The particular number representation is also referred to as "signed-digit number representation".

The normal carry propagation delay is overcome by the underlying carry-free operation of redundant number systems, which is achieved through limiting carry propagation to one position to the left during addition or subtraction. The addition time for such SD numbers of any length is equal to the time required for adding only two adjacent digits, and thus is practically independent of the total world length. However, the transfer digit between adjacent digits might assume both positive and negative values in SD operation and will never propagate past the first adder position on the left.

Meanwhile, redundant number arithmetic can also be used in conventional least-significant-digit (LSD) first addition [50, 51] and multiplication [52]. It can take advantage of the equivalence between redundant-to-binary (RB) conversion and binary addition [53].

3.2 Multiplication

3.2.1 Basic Operations

Generally speaking, multiplication can be viewed as repeated shifts and additions, and in an extreme case, it can be achieved through the use of a single set of adder, shift register, and minimal control logic. In most cases, however, multipliers are not implemented this way because it is prohibitively slow due to its low-speed addition and the maximum number of additions performed.

3.2.2 Implementation Schemes

3.2.2.1 Multiple Scan & Shift Multiplication

Scanning more than one multiplier bit per cycle can speed up multiplication if multiple shifts are performed after each addition. An non-overlapped 2-bit scanning multiplier can be constructed using a carry save adder (Figure 3.15 [54]).

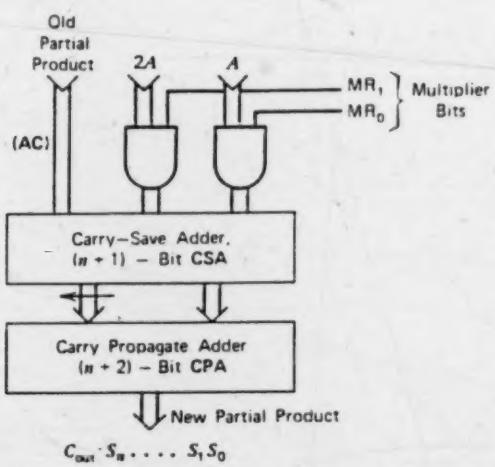


Figure 3.15 Carry save and propagate adder using 2-bit multiplier scan.

In practice, more sophisticated approaches like various recoding schemes were purposed for handling more than one bit per cycle, and they can be realized in both serial and parallel forms.

3.2.2.2 String Recoding and Booth Multiplie

The underlying principle of string recoding is illustrated by Equation 3.16 (i, k are integers), in which the number of additions can be effectively reduced through shifting across a string of zeros in the operand:

$$2^{i+k-1} + 2^{i+k-2} + \ldots + 2^{i+1} + 2^{i} = 2^{i+k} - 2^{i} \qquad (3.16)$$

This observation is normally utilized through grouping and recoding several adjacent bits using signed-digits (SD). It is attractive for converting binary vectors containing long sequences of ones, while binary vectors with many isolated ones deteriorate the performance of the scheme.

One realization of this scheme is the Booth recoding multiplier [55, 56, 57]. Radix-4 recoding (also called the "modified Booth recoding", whose algorithm is shown in Table 3.1, in which MR is the multiplier and PP is the partial product), which scans three bits while recoding two of them per cycle, is the most commonly used. Higher radix recoding schemes were also proposed [58], like the radix-8 recoding scheme (Table 3.2).

One implementation of the Booth recoding scheme is shown in Figure 3.16 [54]. This implementation carries out the multiplication of two 2's complement numbers. In the Figure, AC is the partial product storage; MR is the multiplier; A is the multiplicand; CSA is the carry save adder; CPA is the carry propagation adder; C is the carry; and S is the sum.

$MR_{i+1,i}$	MR_{i-1}	Action
00	0	Shift PP 2 places.
00	1	Add MD, shift PP 2 places.
01	0	Add MD, shift PP 2 places.
01	1	Add 2xMD, shift PP 2 places.
10	0	Subtract 2xMD, shift PP 2 places.
10	1	Subtract MD, shift PP 2 places.
11	0	Subtract MD, shift PP 2 places.
11	1	Shift PP 2 places.

Table 3.1 Radix-4 Booth Recoding Algorithm.

$MR_{i+2,i+1,i}$	MR_{i-1}	Action
000	0	Shift PP 3 places.
000	1	Add MD, shift PP 3 places.
001	0	Add MD, shift PP 3 places.
001	1	Add 2xMD, shift PP 3 places.
010	0	Add 2xMD, shift PP 3 places.
010	1	Add 3xMD, shift PP 3 places.
011	0	Add 3xMD, shift PP 3 places.
011	1	Add 4xMD, shift PP 3 places.
100	0	Subtract 4xMD, shift PP 3 places.
100	1	Subtract 3xMD, shift PP 3 places.
101	0	Subtract 3xMD, shift PP 3 places.
101	1	Subtract 2xMD, shift PP 3 places.
110	0	Subtract 2xMD, shift PP 3 places.
110	1	Subtract MD, shift PP 3 places.
111	0	Subtract MD, shift PP 3 places.
111	1	Shift PP 3 places.

Table 3.2 Radix-8 Booth Recoding Algorithm.

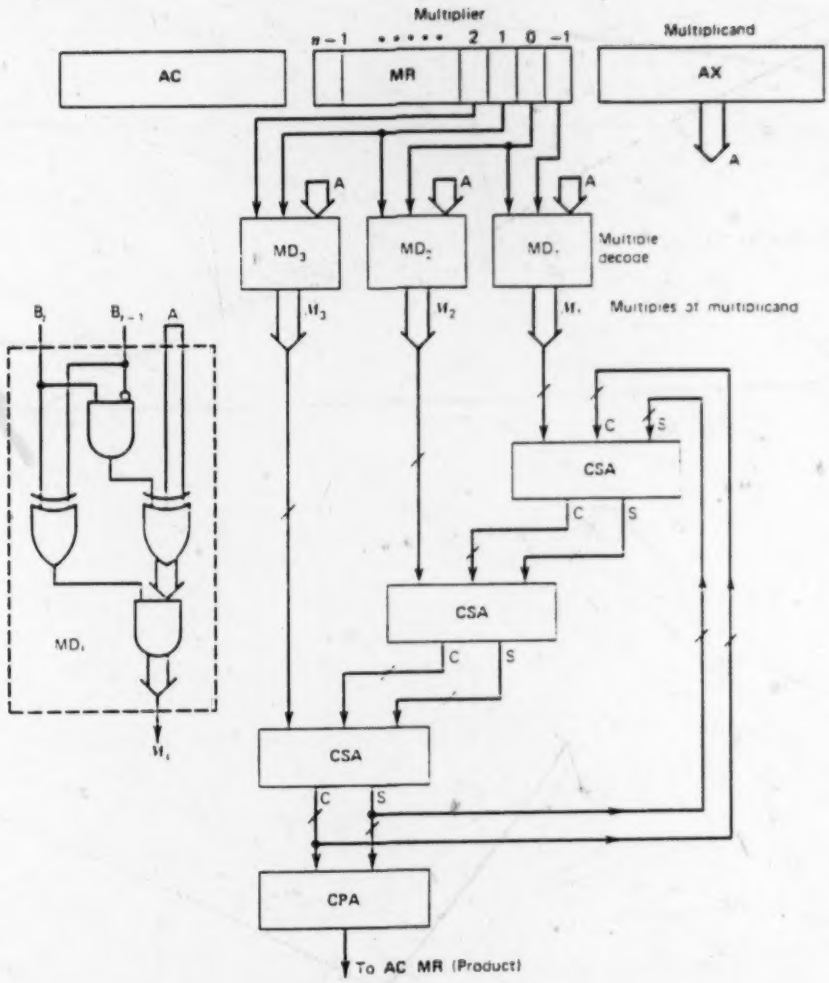


Figure 3.16 Schematic of a 4-bit scanning Booth multiplier using CSA/CPA.

Practical implementations of the Booth recoding scheme normally yield a high performance at relatively low power-consumption. One concern is possible glitches during operation, and a method addressing them was reported [59].

3.2.2.3 Canonical Recoding

Another recoding scheme is called "canonical recoding" [54], in which non-zero digits are separated by zero(s). Its adoption can result in more shifts and less addition. A simple canonical recoding algorithm is shown in Table 3.3. Higher radix recoding algorithms are also available.

Input: $MR_{i+1,i}$	Carry-in: C_i	Recoded: D_i	Carry-out: C_{i+1}
00	0	0	0
01	0	1	0
10	0	0	0
11	0	-1	1
00	1	1	0
01	1	0	1
10	1	-1	1
11	1	0	1

Table 3.3 Radix-4 Canonical Multiplier Recoding Algorithm.

Due to the demand for high-performance multipliers and the prevalence of VLSI technology, the design of high-speed cellular array multipliers is justified and can be viewed as another example of trading hardware investment for performance.

3.2.2.4 Array Multiplier and its Modification

The advantages of array multipliers lie in their regular structure and local interconnects, which translate into easier and more efficient layout. Moreover, their performance can be

further improved by incorporating a high-speed adder during the last stage (Figure 3.17 [41]). In a modified array multiplier, the last level can be implemented using CPA. Also, CSAs can be used to implement full-adders.

However, a study [60] showed that almost 50% of the power was lost due to spurious transitions of internal nodes before they settled down to their final values, a problem resulting from non-uniform path delays. These spurious transitions can be reduced by equalizing path delays from inputs to outputs using latches and/or self-timed techniques using replicated circuit blocks. New array topology that reduces waits between signals at various intermediate stages was suggested, and results in higher speed and lower power dissipation [61].

An algorithm for designing array multipliers using multiplexers was reported [62], which permits efficient VLSI realization, yet achieves high performance.

3.2.2.5 Wallace Tree/Dadda Multiplier

The principle behind Wallace tree multipliers [63] is to create maximum concurrency among their carry save adders. The scheme is shown in Figure 3.18 [41]. Every carry save adder at each stage takes in three operands and produces two results, reducing the number of outputs (inputs to the next stage) by a factor of 3/2. The total number of stages needed is calculated to be $[\log_{1.5} N/2]$. Final addition is carried out when there are only two outputs left, and by a carry propagate adder (2N-bit wide for NxN multiplication).

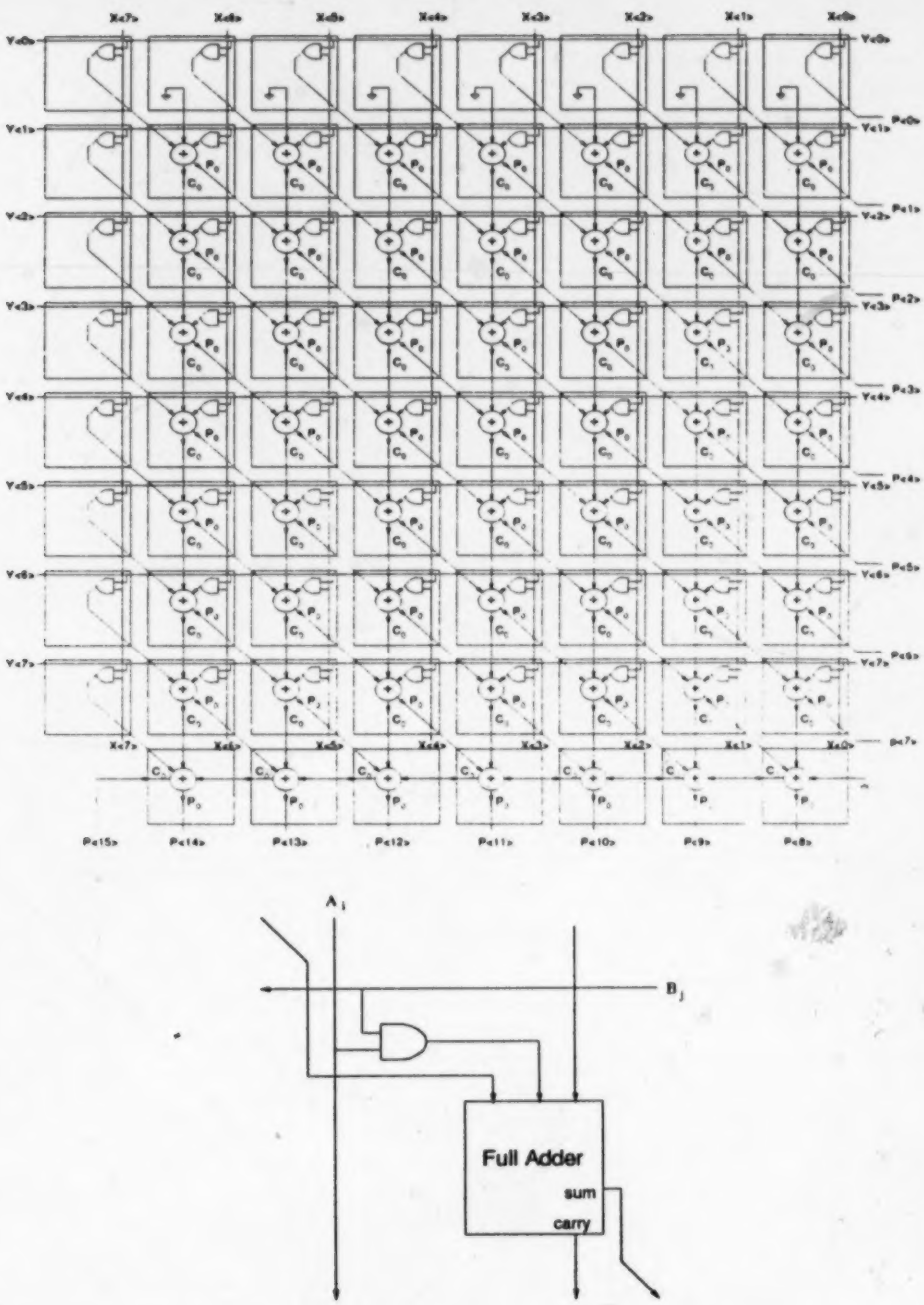


Figure 3.17 Schematic of an 8-bit array multiplier with a unit cell.

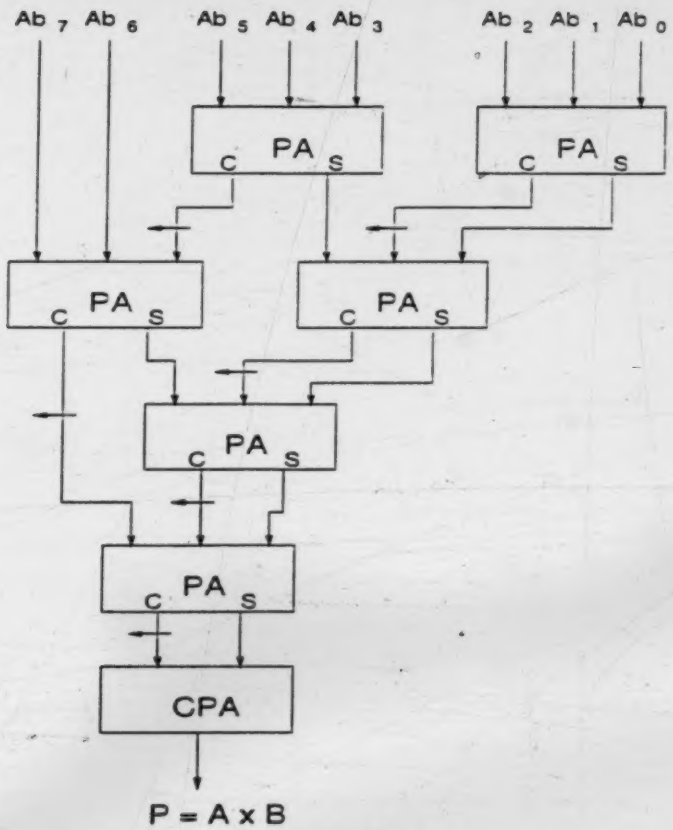


Figure 3.18 Schematic of an 8-bit Wallace tree multiplier.

Generally speaking, a Wallace tree multiplier has fewer stages than a parallel array multiplier, given the same operand width. As a result, it generally yields a better performance, assuming no pipelining [64]. Also, it is arguably more energy-efficient due to the parallel summation of partial products (PPs) [65]. However, these improvements in performance come with a price: Wallace tree multipliers require wider carry propagate adder during the final stage, and adder connection patterns are much more complex (irregular) than those in parallel array multipliers, a factor translating into the need for a larger area and more efforts for VLSI implementation.

Dadda modified the Wallace tree by noting that a full adder can be viewed as a counter of ones in input, and which then outputs the amount in binary form [66]. The height of the matrix at each stage could be maximally reduced by a factor of 1.5, and the following sequence can serve as a reference for reducing the partial product matrix to the final two rows to be added using a carry propagate adder. An 8-bit multiplier is shown in Figure 3.19 [41].

$$\{2, 3, 4, 6, 9, 13, 19, 28, 42, 63, \ldots\} \tag{3.17}$$

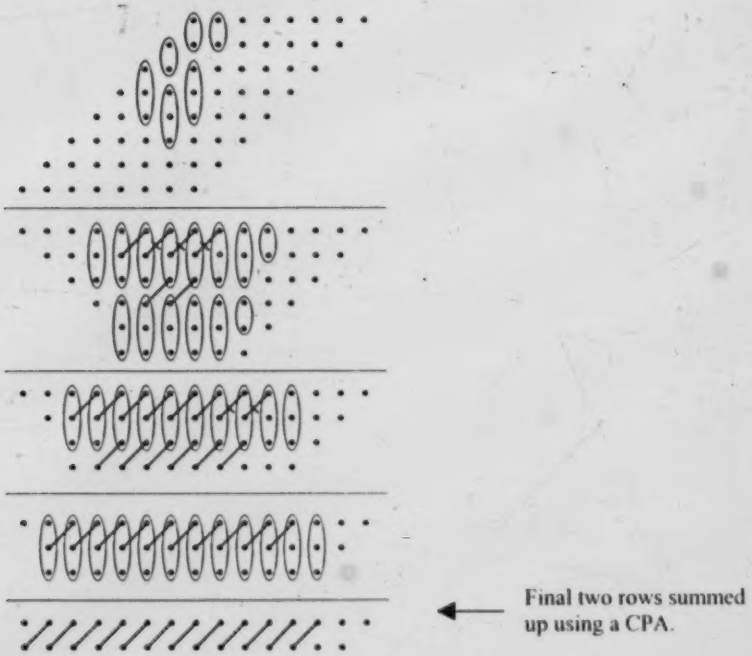


Figure 3.19 Reduction scheme of an 8-bit Dadda multiplier.

In the Figure, each dot represents a bit in the partial product. Ovals indicate half or full adders. Lines into the next level indicate sum and carry out of adders. Crossed lines indicate half adder outputs. The two rows in the last section are summed up using a CPA.

Compared with the Wallace tree, they have the same worst case delay, while Dadda's method generally requires fewer full and half adders, but a slightly wider CPA. This benefit comes at the expense of an even less regular implementation.

3.2.2.6 Digit-Serial Multiplier

As introduced in the addition section, digit-serial implementation is used mainly for signal processing applications requiring moderate sampling rates.

Digit-serial multiplier design was reported [67], as well as its design methodology [68]; the research claimed achieving reasonable performance and low power operation.

3.2.2.7 Other Array Multipliers

Other schemes for array multiplication have been proposed, including (i) Pezaris array multiplier [69] and its modifications, which use a mixture of different types of full adders to carry out direct two's complement array multiplication; and (ii) Baugh-Wooley two's complement multiplier [70], which uses conventional Type-0 full adders to construct and achieve a uniform array structure.

Besides the above-mentioned rather conventional multiplication schemes, other designs were also reported.

3.2.2.8 ROM-Adder Multiplication Networks [71]

A lookup table can also be used for fast multiplication taking advantage of the availability of inexpensive ROM. The results of all possible combinations of operands can be stored in ROM, and the m-bit multiplicand and n-bit multiplier are combined to

define a unique address in the memory. If no rounding is allowed, the capacity required for the ROM is

$$N = 2^{m+n} \times (m+n) \text{ bits} \quad (3.18)$$

When exceedingly large ROM is to be avoided, the product can be rounded to p bits, and the required capacity becomes

$$N = 2^{m+n} \times p \text{ bits} \quad (3.19)$$

3.2.2.9 Logarithmic Multiplication [72]

Another method of reducing the required ROM capacity is through converting multiplication/division into addition/subtraction by means of logarithm and antilogarithm transformations. Thus, instead of storing the whole table, one need to store only the much smaller logarithm, antilogarithm, and addition tables. However, this scheme has inherent error problems, and slower speed due to more ROM accesses.

Other schemes have been proposed, including no bit-slice summing tree multiplication and recoded array multiplication [73].

3.3 Circuit Design Styles

Besides various addition and multiplication schemes, circuit design styles are also being intensively investigated in search of a better compromise among performance, area, and power consumption. In this respect, most attention is paid to full adder design, not only because addition is the deciding factor for performance and power consumption in normal multipliers, but also because multiplier design involves more complex tradeoffs.

3.3.1 Static CMOS Logic

A full adder can be implemented using static CMOS logic, made up of both P and NMOS trees [25]. For this style, both true and complemented inputs, sum, and carry are required or produced, while the sum and carry are computed independent of each other.

3.3.2 CMOS Transmission Gate Logic

The XOR logic and thus the adder can be realized using transmission gates [25]. One noteworthy thing about this implementation is the equal sum and carry delay times.

3.3.3 Complementary Pass-Transistor Logic and Double Pass-Transistor Logic

Complementary Pass-Transistor Logic (CPL) and Double Pass-Transistor Logic (DPL) are examples of logic families designed for high performance, yet low power operation [55, 74]. However, logic styles like CPL, which generates threshold voltage drops at any point in the circuit, are dangerous for low-voltage operation, even when level restoration logic is used. Different opinions exist about the comparison between CMOS and CPL in terms of performance and power [75, 76]. DPL is claimed to be fast and energy-efficient [74, 77, 78]. The problem with DPL arises when layout issues are considered. The differential nature of DPL and the limited ability to exploit source-drain sharing in layout make DPL less efficient than CMOS.

3.3.4 Cascode Voltage Switch Logic (CVSL)

CVSL [41] belongs to the dynamic logic family and requires a two-phase clock with no complementary clock signal. For full adder operation, the outputs and their complements

are all pre-charged high while the clock is low. When the clock goes high, the complementary cascoded differential NMOS tree pulls either the output or its complement low.

3.3.5 Differential Cascode Voltage Switch Logic (DCVS)

DCVS can be constructed as a static version of CVSL by replacing the PMOS with a cross-coupled pair of PMOS. DCVS yields very fast operation in terms of worst case delay [41].

3.3.6 Other circuit design styles

Other circuit design styles include [41] No Race Dynamic CMOS Logic (NORA), CMOS Non-Threshold Logic (CNTL), and Enable/disable CMOS Differential Logic (ECDL), as well as several others proposed for low-power operation [79, 80].

3.4 Characterization

Several commonly used criteria characterize designs in terms of area, delay, and power consumption.

Area usage is normally estimated through the number of transistors used. Delays are measured through worst case delay between input and the slowest output, typically the most significant sum bit. Power consumption is often approximated through (i) peak switching current, which is defined as the largest difference between the steady state

current of the power supply and the power supply current during computation, and (ii) average power dissipation.

Commonly used test strategies can normally be divided into logic simulation, switch-level simulation and physical measurement.

3.5 Conclusions

This chapter presented a comprehensive review of various fixed-number addition and multiplication designs, highlighting their underlying principles, realization methods, as well as tradeoffs between performance, area usage and power consumption.

Among addition designs, carry lookahead adders are the fastest. Carry select adders are also fast, but they achieve high performance with area and power overhead. Carry save adders achieve reasonable speed at relatively lower power consumption. They are selectively used in most high-performance DSP systems.

Among multiplication designs, array multipliers used to be adopted because of their ease of layout. Wallace tree multipliers are commonly used now due to their high speed. When power consumption is a concern, Booth recoding multipliers are normally selected. Since the multiplication components in baseband DSP systems need to satisfy high throughput, small area usage, and low power-consumption requirements, it will be desirable for these components to adopt Booth recoding schemes, incorporate various architectural level techniques like pipelining and parallelism, as well as use suitable addition designs within certain pipeline stages. Based on this study, a novel retargetable, high throughput yet power-efficient multiplication component has been developed.

Chapter 4

Retargetable Arithmetic Architecture for Low-Power Baseband DSP Supporting the Design for Reusability Methodology

4.1 Introduction

Due to the rapid advances in VLSI fabrication technology, it is now feasible to implement complete systems on single integrated circuits. This approach of doing hardware design, also called "System-on-a-Chip" (SoC), is gaining increasing acceptance and support in the semiconductor industry. Hardware designers are now facing system designs with not only tens of millions of transistors, but also of unprecedented complexity. These factors, when coupled with the ever-pending time-to-market pressure, and the stringent requirements for performance and power-efficiency, makes developing contemporary systems from scratch virtually impossible. Thus, to design reusable

component cores (macros) and incorporate them in SoC designs becomes a promising approach for bridging the gap between available gate-count and designer productivity. However, effective use of this core-based design approach for SoC designs requires an extensive library of reusable pre-designed and pre-verified cores, which are created following a consistent design methodology.

Aiming at possible incorporation in the baseband DSP system, a reusable SoC multiplication component was developed during this thesis research. Its features, structures, design methodology, verification results, and design-for-testability considerations are discussed in the following sections.

4.2 Features of the Reusable Multiplication Component

Based on the study and understanding of underlying principles and past work discussed in Chapters 2 and 3, a novel reusable multiplication component was developed focusing on providing the following features: (i) retargetability, (ii) high throughput, and (iii) low-power operation. As well, the component takes the form of a reusable soft core. Retargetability was demonstrated through the component's ability to switch between radix-4 and 8 recoding schemes for carrying out multiplication, and its capability to handle different operand lengths. In this project, 8 and 12-bit two's complement operands were considered for the module, with the two operands accepted simultaneously being of the same length. The choices for different radix recoding schemes and operand widths were sent to the component through two control signal bits. The calculation results were also in two's complement form. This retargetability was successfully obtained and

verified through exhaustive simulation. Other features were achieved through the incorporation of architectural level design styles like pipelining, parallelism and gated circuits. The estimated throughput after synthesis is 171.5 MIPS, and this component will need at least 52.6% fewer addition operations than those required by a conventional multiplier, i.e., an array multiplier. All these results fulfill the research objectives stated in Section 1.3.

4.3 Structures of the Reusable Multiplication Component

4.3.1 Overview

This novel retargetable multiplication component has three main sections (Figure 4.1). The recoding section transforms the multiplicand into coded operands based on slices of the multiplier and the recoding scheme adopted. The number of coded operands produced varies, depending on the word length of the multiplicand and multiplier, as well as the recoding scheme, with six being the maximum number when the original operands are 12-bit, and redix-4 recoding is adopted. This recoding section is further divided into four pipelining stages, along with other architectural level techniques for higher throughput and power efficiency.

The first custom addition section adds together two adjacent coded operands from the preceeding recoding section and produces up to three partial products. In this section, carry-save, carry lookahead, and carry select addition schemes are used, while on the architectural level, pipelining, parallelism and methods for reducing circuit switching are utilized and will be explained in more detail later this section.

The second custom addition section adds together up to three partial products from the preceding custom addition section and produces the final result of multiplication. In this section, carry-save, carry-lookahead, and carry select addition schemes are also used, though taking a different form from the first custom addition section, along with architectural level techniques.

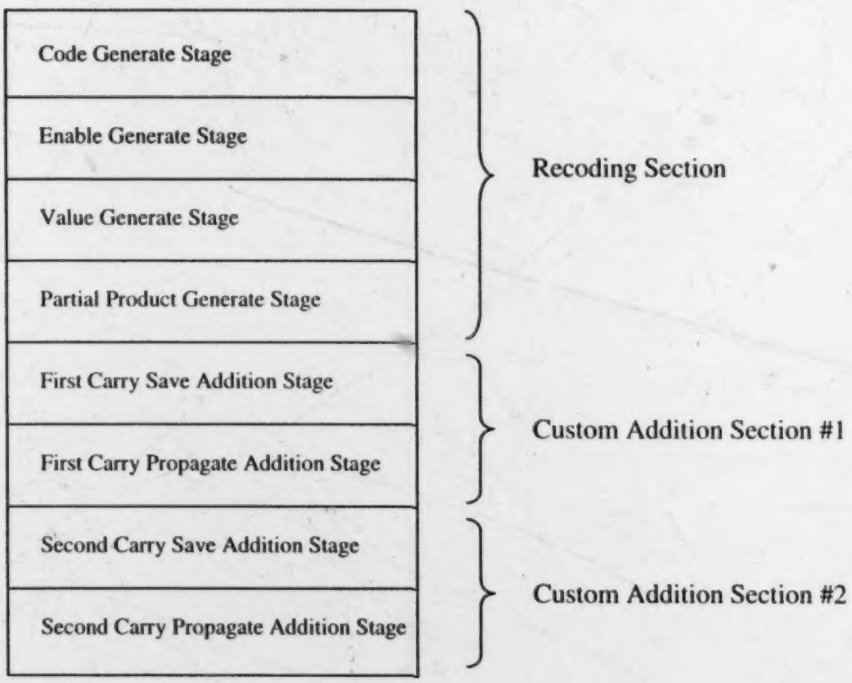


Figure 4.1 Illustrative diagram of main structures of the multiplication component.

Control signals of this multiplication component are made up of two bit, namely, sel_rad and sel_bit, with their function illustrated in Table 4.1.

sel_rad	sel_bit	Selected Mode
0	0	Accepts 8-bit operands, and uses Rad-4 recoding scheme.
0	1	Accepts 12-bit operands, and uses Rad-4 recoding scheme.
1	0	Accepts 8-bit operands, and uses Rad-8 recoding scheme.
1	1	Accepts 12-bit operands, and uses Rad-8 recoding scheme.

Table 4.1 Function of the control signal.

4.3.2 Recoding Section

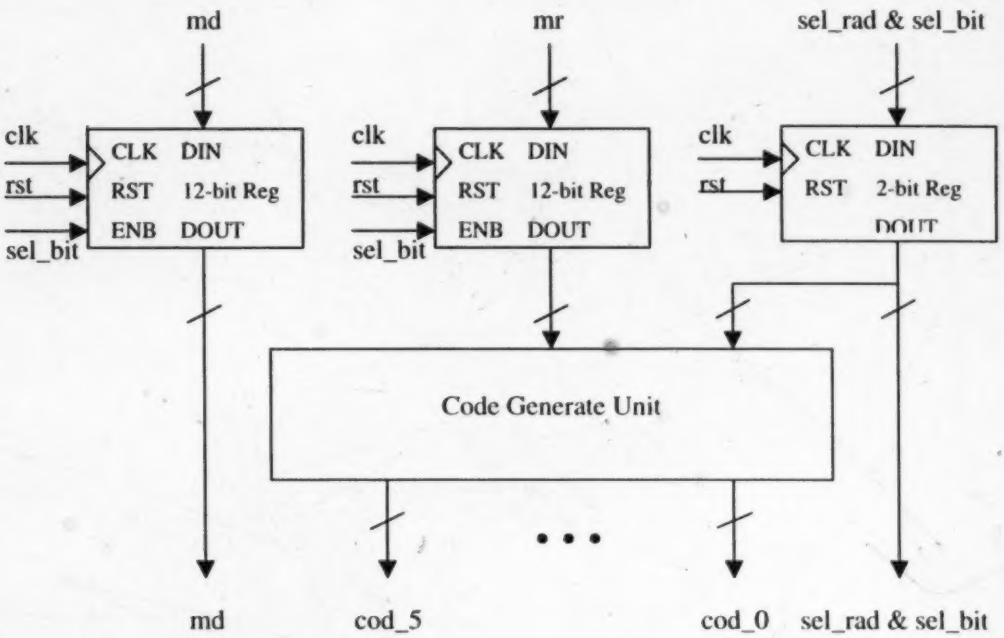
As mentioned in the overview of this multiplication component, the recoding section is further divided into four pipelining stages.

4.3.2.1 Code Generate Stage

The code generate stage (Figure 4.2) is the first pipeline stage in the recoding section. It consists of two 12-bit registers, one 2-bit register, and one code generate unit.

The registers accept and store both operands and control signals, and synchronize the operation of this stage. The functionality of this stage is mainly carried out in the code generate unit. Basically, it divides the multiplier (mr) into three to six 2-bit or 3-bit wide slices, depending on the recoding scheme selected. When radix-4 recoding scheme is chosen, three digits of the multiplier are scanned at a time, with two of them recoded

based on the rule prescribed in Table 3.1. On the other hand, when the radix-8 recoding scheme is chosen, four digits of the multiplier are scanned at a time, with three of them recoded based on the rule prescribed in Table 3.2. Generated codes indicate the partial products that need to be produced (Table 4.2). The structure of the code generate stage can also be captured in the form of the VHDL code (Appendix 1).



md: *multiplicand*

mr: *multiplier*

sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*

sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*

clk: *clock signal*

rst: *reset signal*

cod_i: *code indicating the partial product that needs to be produced, i= 0,1, 2, 3, 4, 5*

Figure 4.2 Code generate stage.

Partial Product Needed	cod_i
1 x md	0001
-1 x md	1111
2 x md	0010
-2 x md	1110
3 x md	0011
-3 x md	1101
4 x md	0100
-4 x md	1100

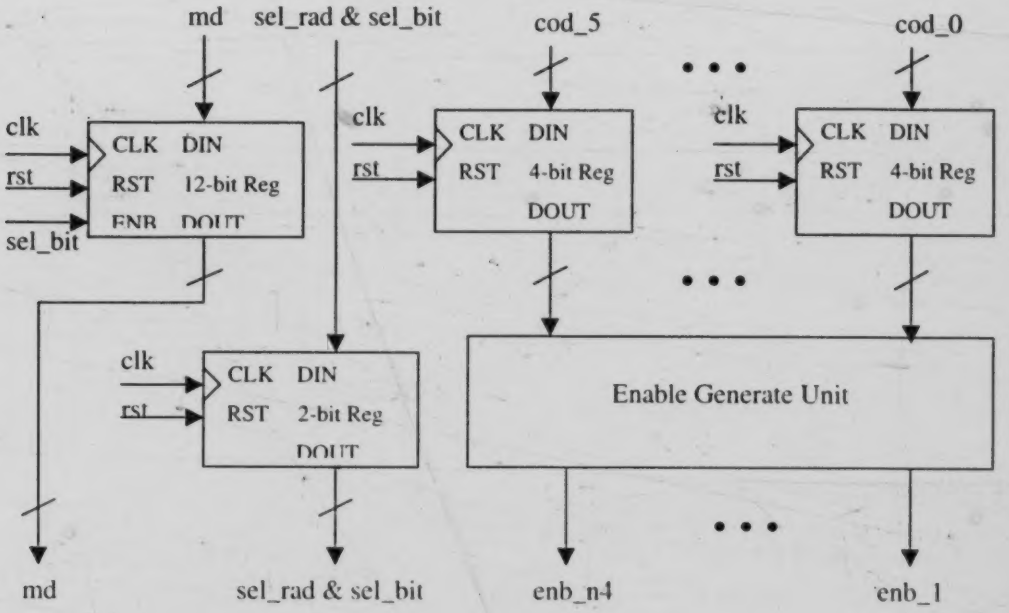
Table 4.2 Correspondence between cod_i and the partial product needed.

4.3.2.2 Enable Generate Stage

The enable generate stage (Figure 4.3) is the second pipeline stage in the recoding section. It consists of one 12-bit register, one 2-bit register, six 4-bit registers, and one enable generate unit.

The registers accept and store the multiplicand, control signals, six 4-bit codes generated from the preceding stage, and synchronize the operation of this stage. This stage's functionality is carried out mainly in the enable generate unit. It accepts the six 4-bit codes (cod_5, cod_4, cod_3, cod_2, cod_1, cod_0) and produces eight enable signals, namely, enb_n4, enb_4, enb_n3, enb_3, enb_n2, enb_2, enb_n1 and enb_1, which indicate that –4•md, 4•md, -3•md, 3•md, -2•md, 2•md, -1•md, and 1•md, respectively,

are needed later in the calculation. The structure of the enable generate stage can also be captured in the form of the VHDL code (Appendix 2).



md: *multiplicand*

sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*

sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*

clk: *clock signal*

rst: *reset signal*

cod_i: *code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*

enb_i: *code indicating that i•md is needed later in the calculation, i = 1, n1, 2, n2, 3, n3, 4, n4*

Figure 4.3 Enable generate stage.

4.3.2.3 Value Generate Stage

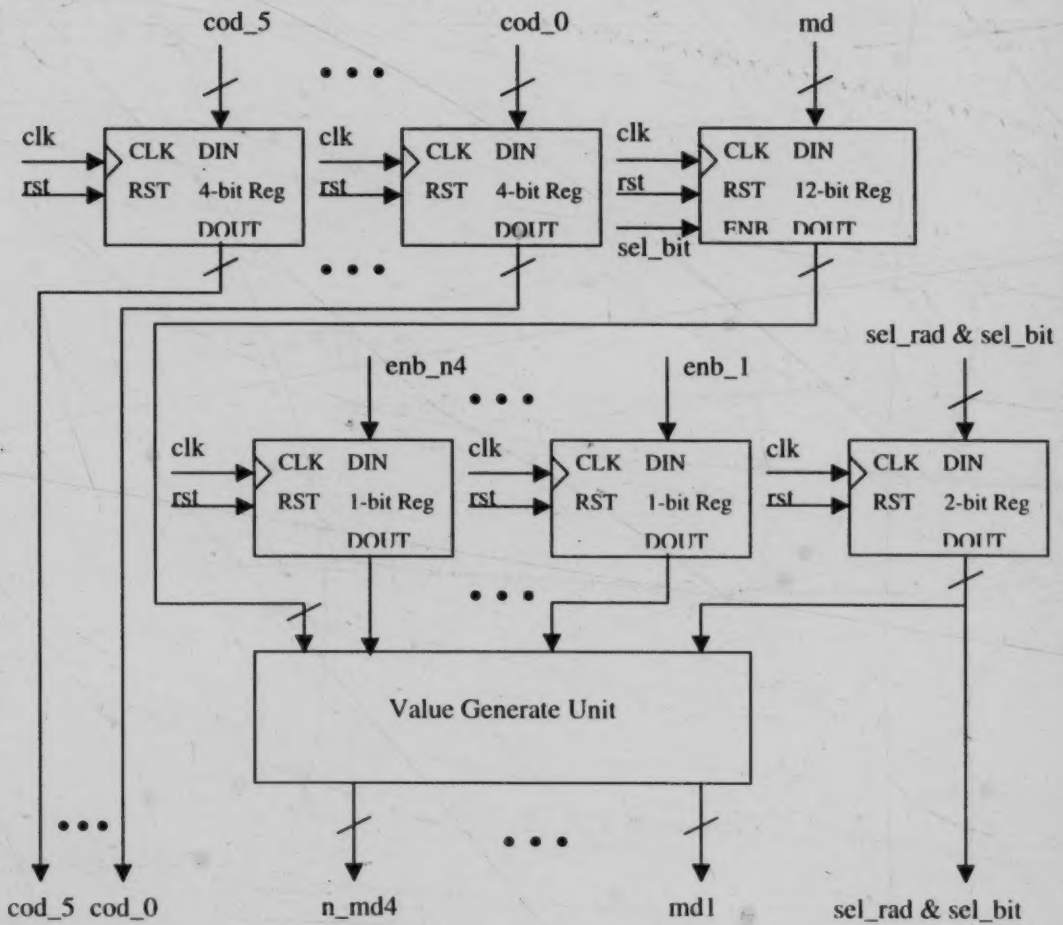
The value generate stage (Figure 4.4) is the third pipeline stage in the recoding section. It consists of one 12-bit register, one 2-bit register, six 4-bit registers, eight 1-bit registers, and one value generate unit.

The registers accept and store the multiplicand, control signals, six 4-bit codes, eight 1-bit enable signals, and synchronize the operation of this stage. This stage's functionality is carried out mainly in the value generate unit. It generates the values of –4•md, 4•md, -3•md, 3•md, -2•md, 2•md, -1•md, and 1•md if the respective enable signal passed on from the preceding stage is asserted. Otherwise, the value is not calculated and generated. The structure of the value generate stage can also be captured in the form of the VHDL code (Appendix 3).

4.3.2.4 Partial Product Generate Stage

The partial product generate stage (Figure 4.5), is the fourth and last pipeline stage in the recoding section. It consists of one 2-bit register, six 4-bit registers, eight 14-bit registers, and one partial product generate unit.

The registers accept and store control signals, six 4-bit codes, eight 14-bit multiplicand products, and synchronize the operation of this stage. This stage's functionality is carried out mainly in the partial product generate unit. It generates the value of the partial product corresponding to each slice of the multiplier. Basically, for each partial product, the unit selects one value among –4•md, 4•md, -3•md, 3•md, -2•md, 2•md, -1•md, and 1•md according to the code generated in the first stage. The structure of this stage can also be captured in the form of the VHDL code (Appendix 4).



md: *multiplicand*

sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*

sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*

clk: *clock signal*

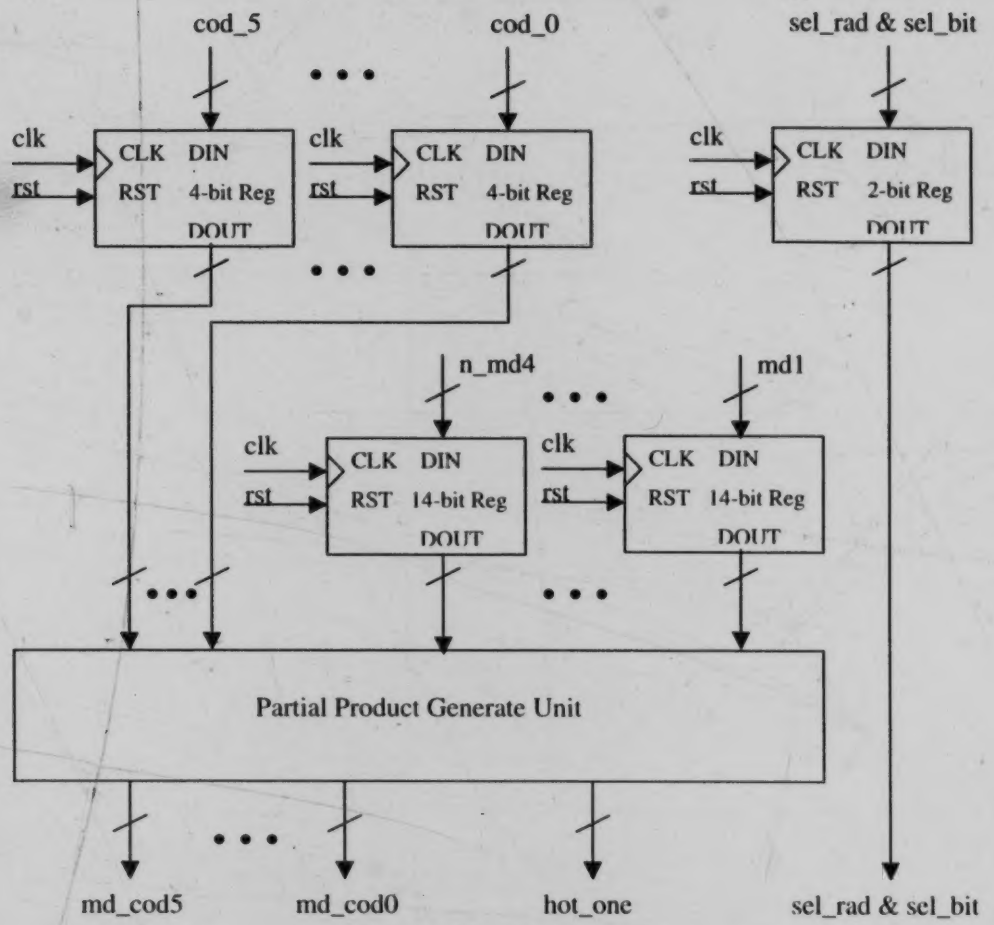
rst: *reset signal*

cod_i: *code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*

enb_i: *code indicating that i•md is needed later in the calculation, i = 1, n1, 2, n2, 3, n3, 4, n4*

(n_)mdi: *value of (-)i•md, i = 1, 2, 3, 4*

Figure 4.4 Value generate stage.



sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*

sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*

clk: *clock signal*

rst: *reset signal*

cod_i: *code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*

(n_)mdi: *value of (-)i•md, i = 1, 2, 3, 4*

md_codi: *ith partial product, i = 0, 1, 2, 3, 4, 5*

hot_one: *code having bitwise correspondence with the sign of each partial product*

Figure 4.5 Partial product generate stage.

4.3.3 Custom Addition Section #1

Immediately after the recoding section is the first custom addition section, which is divided into two pipeline stages.

4.3.3.1 First Carry Save Addition Stage

The first pipeline stage in the first custom addition section is a carry save addition (CSA) stage (Figure 4.6). It consists of one 2-bit register, two 13-bit registers, four 14-bit registers, one 6-bit register, one align and ready unit, and three CSA units.

The registers accept and store control signals, two 13-bit and four 14-bit partial products, one 6-bit sign code, and synchronize the operation of this stage. After each pair of adjacent partial products is aligned and becomes ready for addition along with the corresponding hot-ones through the align and ready unit, the pair is fed into three custom-designed carry save addition units for parallel addition. Each addition unit handles two adjacent partial products as well as the hot-ones associated with them, and produces one set of partial sum (result1_i) and carry-out (cout1_i) for use in the following carry propagate addition stage. The structure of the first carry save addition stage can also be captured in the form of the VHDL code (Appendix 5).

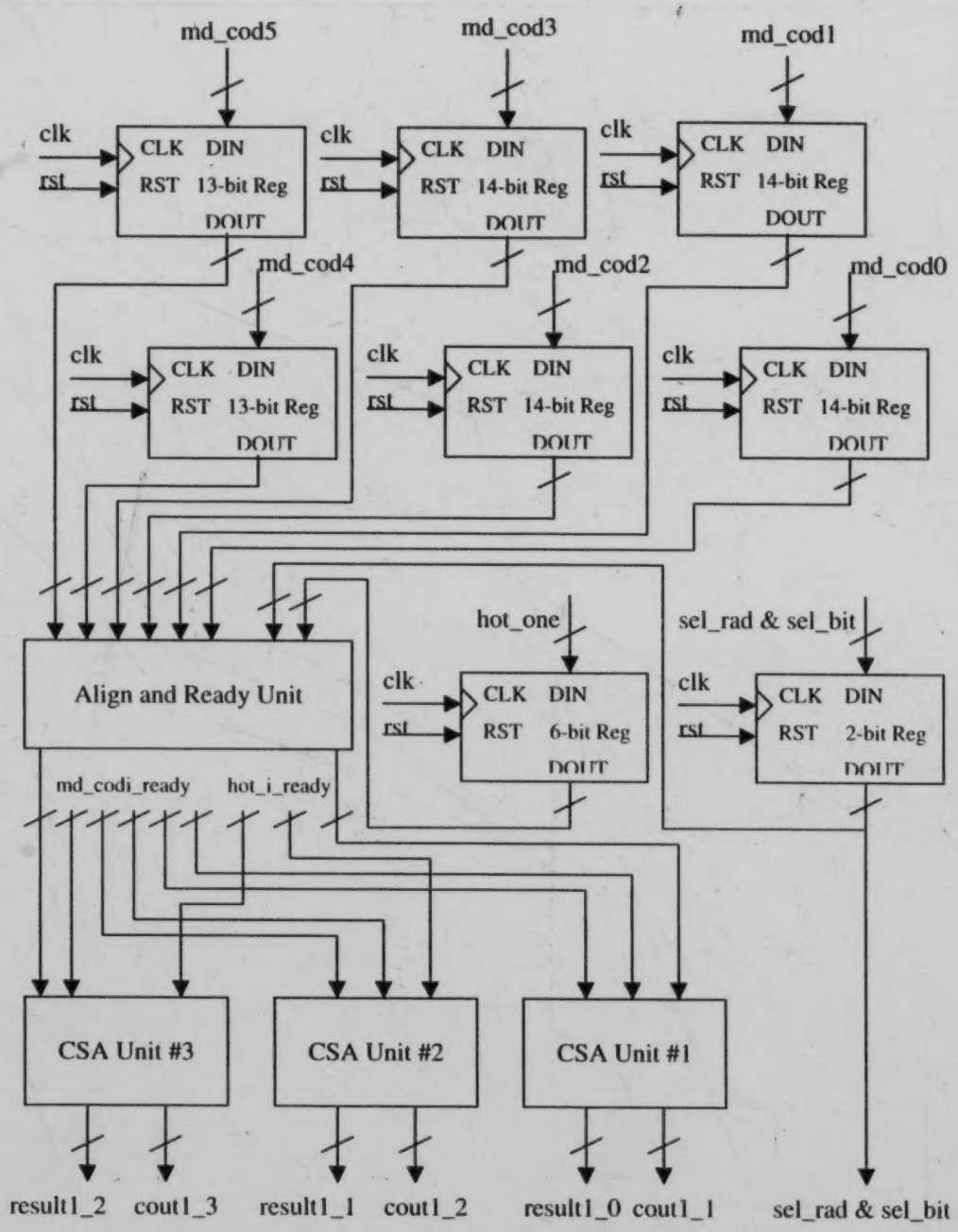


Figure 4.6 First carry save addition stage (cont'd on the next page).

sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*
sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*
clk: *clock signal*
rst: *reset signal*
md_codi: *ith partial product, i = 0, 1, 2, 3, 4, 5*
hot_one: *code having bitwise correspondence with the sign of each partial product*
result1_i: *partial sum, i = 0, 1, 2*
cout1_i: *carry-out, i = 1, 2, 3*

Figure 4.6 (cont'd) First carry save addition stage.

4.3.3.2 First Carry Propagate Addition Stage

The other pipeline stage in the first custom addition section is a carry propagate addition (CPA) stage (Figure 4.7). It consists of one 2-bit register, one 15-bit register (expanded to 16-bit for synthesis), one 16-bit register, two 17-bit registers (expanded to 18-bit for synthesis), two 18-bit registers, three CPA units, and one post-addition unit.

The registers accept and store control signals, three partial sums, three carry-outs, and synchronize the operation of this stage. There are three custom carry propagate addition units in this stage. Each accepts a pair of partial sum and carry-out produced in the preceding carry save addition stage and adds them together to provide an intermediate result (result2_i). As before, these additions are carried out in parallel. Within each carry propagate addition unit, carry lookahead and carry select addition schemes (both being described in Section 3.1.2) are adopted in order to speed up the calculation. The structure of the first carry propagate addition stage can also be captured in the form of the VHDL code (Appendix 6).

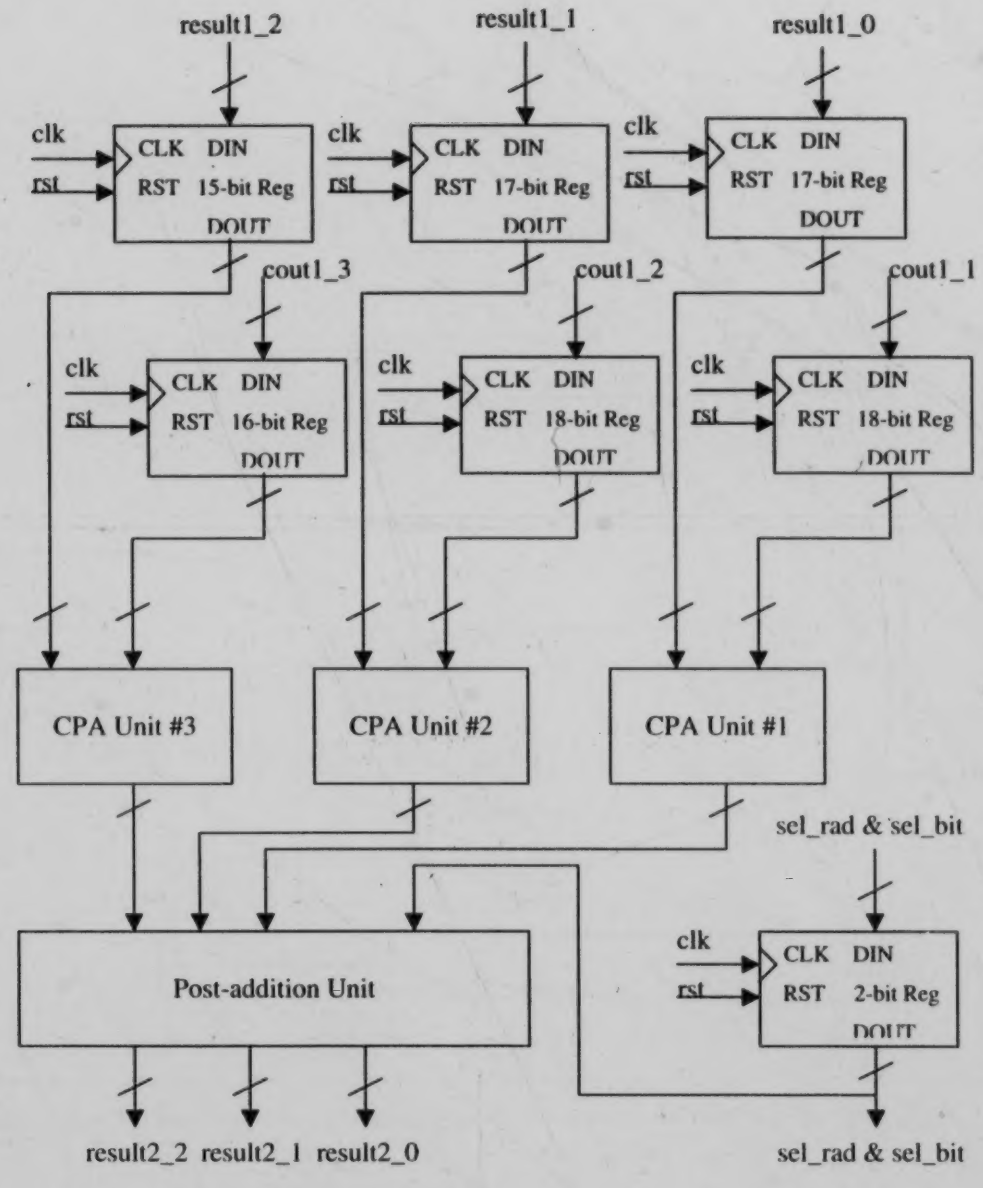


Figure 4.7 First carry propagate addition stage (cont'd on the next page).

sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*
sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*
clk: *clock signal*
rst: *reset signal*
result1_i: *partial sum, i = 0, 1, 2*
cout1_i: *carry-out, i = 1, 2, 3*
result2_i: *intermediate result, i = 0, 1, 2*

Figure 4.7 (cont'd) First carry propagate addition stage.

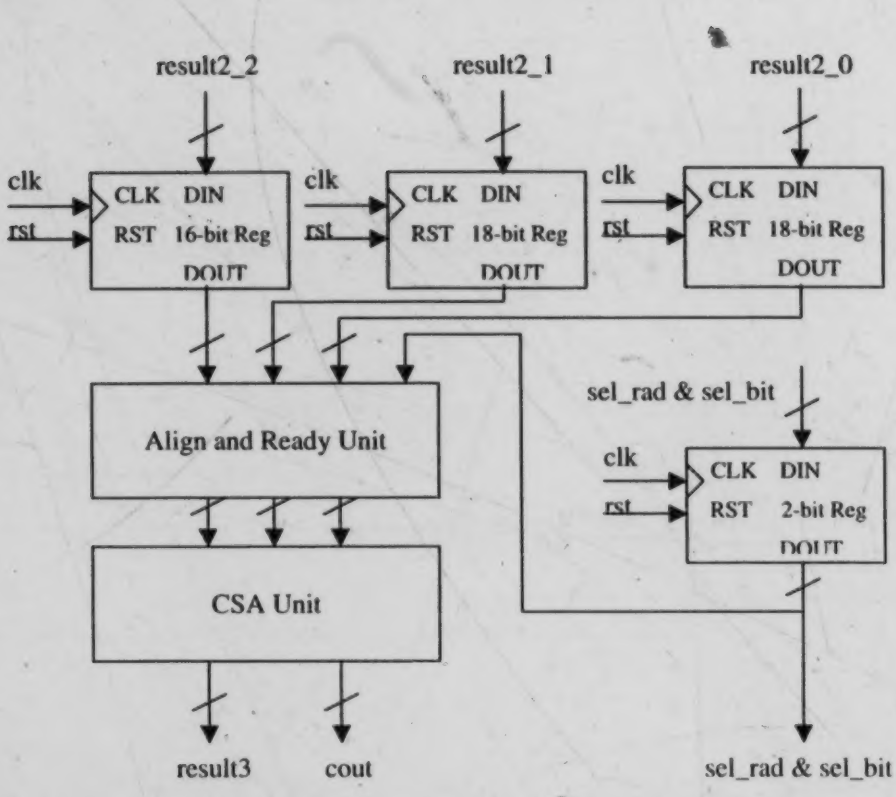
4.3.4 Custom Addition Section #2

The second custom addition section is the final section of the multiplication component and is also divided into two pipeline stages.

4.3.4.1 Second Carry Save Addition Stage

The first pipeline stage in this section is a carry save addition (CSA) stage (Figure 4.8). It consists of one 2-bit register, one 16-bit register, two 18-bit registers, one align and ready unit, and one CSA unit.

The registers accept and store control signals, three intermediate results, and synchronize the operation of this stage. Three intermediate results (result2_i) are provided from custom addition section #1. After being aligned and becoming ready for addition, they are fed into the custom carry save addition unit for addition and produce a partial sum (result3) and carry-out (cout). The structure of the second carry save addition stage can also be captured in the form of the VHDL code (Appendix 7).



sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*

sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*

clk: *clock signal*

rst: *reset signal*

result2_i: *intermediate result, i = 0, 1, 2*

result3: *partial sum*

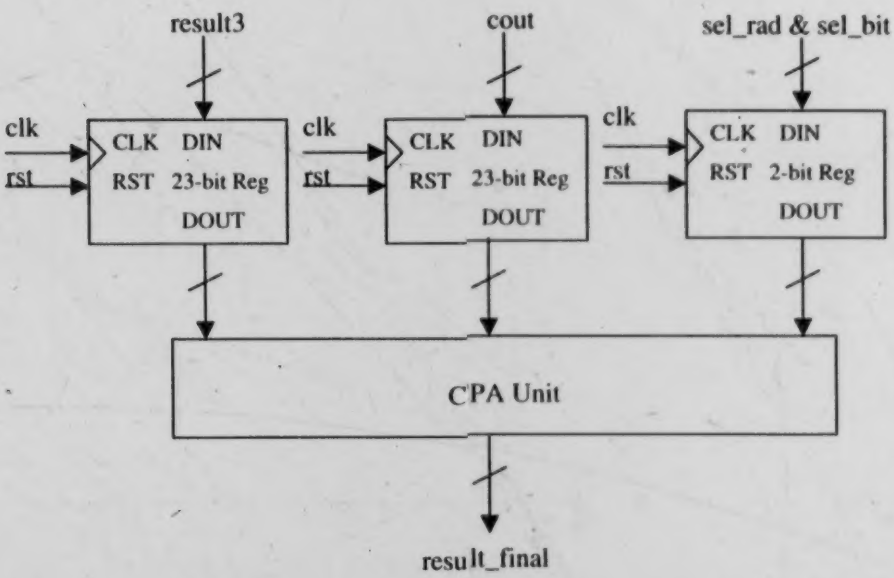
cout: *carry-out*

Figure 4.8 Second carry save addition stage.

4.3.4.2 Second Carry Propagate Addition Stage

The other pipeline stage in the second custom addition section is a carry propagate addition (CPA) stage (Figure 4.9)). It consists of one 2-bit register, two 23-bit registers, and one CPA unit.

The registers accept and store control signals, the partial sum and carrry-out produced in the preceding carry save addition stage, and synchronize the operation of this stage. There is a custom carry propagate addition unit in this stage. It adds the partial sum (result3) and carry-out (cout) together to produce the final result (result_final). Again, carry lookahead and carry select addition schemes (both described in Section 3.1.2) are adopted in order to speed up the calculation. The structure of the second carry propagate addition stage can also be captured in the form of the VHDL code (Appendix 8).



sel_rad: *control signal for selecting radix-4 or radix-8 recoding scheme*
sel_bit: *control signal for selecting 8-bit or 12-bit operand wordlength*
clk: *clock signal*
rst: *reset signal*
result3: *partial sum*
cout: *carry-out*
result_final: *final result*

Figure 4.9 Second carry propagate addition stage.

4.4 Design Methodology

4.4.1 Design Flow

As discussed in this chapter's introduction, a new chip design approach, System-on-a-Chip (SoC), comes into play along with the advances in VLSI fabrication technology. These pose new challenges to hardware designers, that is, while the time-to-market pressure still exists and the design team and tool see no dramatic change, the gate-count and complexity of the chip increase exponentially. Thus, hardware designers must seek possible opportunities for design reuse instead of building everything from scratch each time a new design is undertaken. However, these pre-designed and verified reusable cores need to be developed under a consistent design methodology in order to facilitate their successful integration into system designs.

Figure 4.10 shows the traditional ASIC design flow. This approach is also called the "waterfall model" and minimizes dependency and interaction between design teams. It works well for designs with up to 100k gates and with feature size no finer than 0.5 μm.

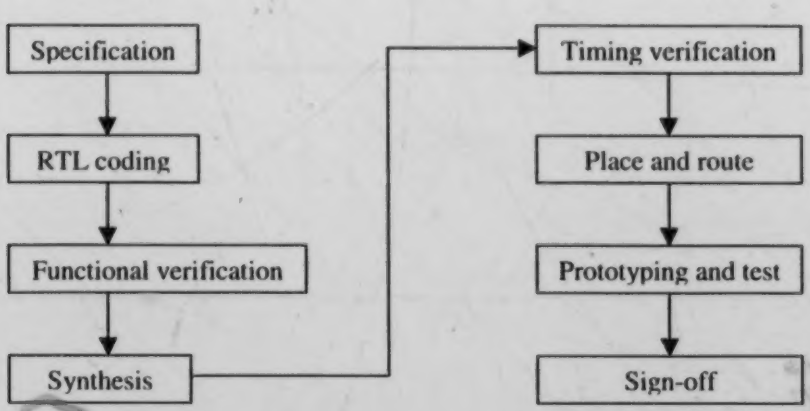


Figure 4.10 Traditional ASIC design flow.

For contemporary designs using state-of-the-art submicron technology and with reuse in mind, modified approaches need to be adopted in order to manage complexity and guarantee first-time success. The methodology adopted for the design of this reusable multiplication component (Figure 4.11), can be considered as a top-down, yet recursive approach.

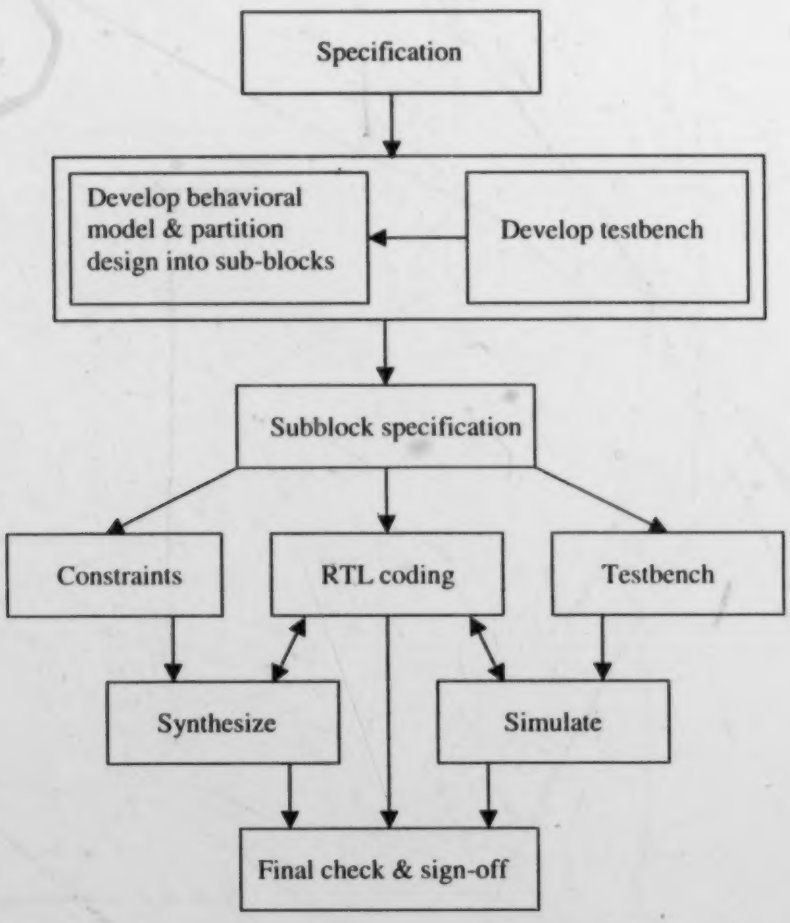


Figure 4.11 Core design flow.

During the development process, the component is handled at different abstraction levels. Testbenches are developed at each level for extensive verification and simulation to ensure that correct functionality and operation are achieved at each level. Also checked are various design constraints like timing and synthesis considerations. If anything turns out to be unsatisfactory, the coding for this block needs to be modified with detrimental effects mitigated and results later re-verified. This cycle ends only when all the functionality and constraint requirements are fulfilled. From another perspective, this whole process can also be considered as "Construct by Correction." This design flow is adopted in order to guarantee that the core has a well understood and stable behavior and can be safely reused and incorporated into a larger system design.

4.4.2 Timing and Synthesis Considerations

Besides design flow, other system-level issues, namely, timing and synthesis considerations, were decided as part of the design methodology.

This multiplication component operates synchronously, with only one clock domain being used for the entire core. Proven to simplify the timing issue, synchronous designs normally result in robust systems and are a common choice for reusable cores. On the other hand, the reset signal is also implemented synchronously because it is straightforward to synthesis since reset becomes another synchronous input to the core. However, a power-up reset is required. All memory elements are implemented using registers to take advantage of the increase in available gate counts and, more importantly, to avoid the time ambiguity associated with latches.

4.5 Results

4.5.1 Retargetability

Exhaustive functional verification of this reusable multiplication component core was carried out, which included (i) modeling of the multiplication behavior, (ii) developing the exhaustive testbench, and (iii) running the simulation.

The VHDL code for multiplier behavior modeling is shown in Appendix 9. In this model two operands are accepted along with the clock and reset signal, and the multiplication result is produced as the only output.

This model and the developed core were combined into the testbench (Appendix 10). The testbench, by itself, generates the clock and reset signals along with the test vectors, and sends them as inputs to both the model and the core. Then, the calculated results from both units were compared. The exhaustive verification approach was adopted in developing the testbench, that is, the generated test vector covered all possible combinations of the multiplicand and multiplier. Another feature of this testbench is its automatic error detection ability, which is illustrated as the following. During the simulation and comparison, if everything is fine, the process will go on until all the test vectors are exhausted; if not, the process will also go on, but a warning message will be shown on the simulator display, indicating a discrepancy between the model and the core, along with the time when this discrepancy occurs. This feature greatly reduces the designer time and energy spent on filtering through long and complex waveforms produced by exhaustive testing.

The simulation was carried out using Mentor Graphics QuickVHDL software. For 8-bit operands, $2^8 x 2^8$ cycles were run for the radix-4 recoding scheme and repeated for the

radix-8 recoding scheme. For 12-bit operands, $2^{12}x2^{12}$ cycles were run for the radix-4 recoding scheme and repeated for the radix-8 recoding scheme. Not until several iterations of simulation were run and extensive debugs were carried out did the developed core show satisfactory performance with retargetability requirements being fulfilled.

4.5.2 Throughput

Each pipeline stage of the developed core was synthesized using Synopsys Design Analyzer software. The basic logic gates used in the synthesis came from the technology library provided by the Canadian Microelectronic Corporation (CMC) called "wcells", which uses the 0.35-micron CMOS (CMOSP35) technology provided by the Taiwan Semiconductor Manufacturing Company Ltd. (TSMC). The model for interconnecting wires (wire load model) used in the synthesis is called "conservative_8k."

The timing part of the synthesis report file for each pipeline stage is included in the following sections. These timing reports show the worst path (maximum delay) in each pipeline stage and are summarized in Table 4.3. Since the core has a pipeline architecture, the maximum delay in each pipeline stage not only decides how fast each stage can run, but also determines the overall core throughput because no correct result can be obtained unless each pipeline stage successfully accomplishes its task. Thus, the throughput of the core can be estimated using the following equation,

$$\text{Throughput} = 1 / \text{Delay}_{max} \qquad (4.1)$$

in which Delay_{max} is the maximum delay of all pipeline stages.

4.5.2.1 Timing Report for Code Generate Stage

Basically, the following timing report (Figure 4.12) shows the worst path (maximum delay) in the code generate stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the outputs in the code generate unit. This observation corresponds to the structure of this pipeline stage (Figure 4.2). Meanwhile, the first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the code generate stage, the maximum delay is 2.27 ns.

```
Startpoint: mr_reg/DOUT_reg[9]
            (rising edge-triggered flip-flop)
Endpoint: COD_3[2] (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
mr_reg/DOUT_reg[9]/ck (wdp_2)                  0.00       0.00 r
mr_reg/DOUT_reg[9]/q (wdp_2)                   0.50       0.50 f
mr_reg/DOUT[9] (REG_OP_0)                      0.00       0.50 f
code_generate/MR_IN[9] (COD_GEN)               0.00       0.50 f
code_generate/U1064/op (winv_2)                0.25       0.75 r
code_generate/U1078/op (wxor2_2)               0.33       1.08 r
code_generate/U1040/op (winv_2)                0.17       1.25 f
code_generate/U997/op (wor2_2)                 0.28       1.54 f
code_generate/U996/op (winv_2)                 0.09       1.63 r
code_generate/U995/op (wnand2_2)               0.13       1.76 f
code_generate/U1048/op (wnand2_2)              0.12       1.88 r
code_generate/U1046/op (wnand2_2)              0.17       2.05 f
code_generate/U1049/op (winv_2)                0.10       2.15 r
code_generate/U1044/op (wnand2_2)              0.12       2.27 f
```

```
code_generate/COD_3[2] (COD_GEN)          0.00       2.27 f
COD_3[2] (out)                            0.00       2.27 f
data arrival time                                    2.27
-------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.12 Timing report for code generate stage.

4.5.2.2 Timing Report for Enable Generate Stage

The following timing report (Figure 4.13) shows the worst path in the enable generate stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the outputs in the enable generate unit. This observation corresponds to the structure of this pipeline stage (Figure 4.3). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the enable generate stage, the maximum delay is 1.44 ns.

```
Startpoint: cod_1_reg/DOUT_reg[1]
            (rising edge-triggered flip-flop)
Endpoint: ENB_1 (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                     Incr       Path
-------------------------------------------------------------
cod_1_reg/DOUT_reg[1]/ck (wdtp_2)         0.00       0.00 r
cod_1_reg/DOUT_reg[1]/q (wdtp_2)          0.65       0.65 f
cod_1_reg/DOUT[1] (REG_N4_4)              0.00       0.65 f
enable_generate/COD_1[1] (ENB_GEN)        0.00       0.65 f
```

```
enable_generate/U149/op (wnor2_2)          0.22       0.87 r
enable_generate/U156/op (wnand3_2)         0.20       1.07 f
enable_generate/U154/op (wnand2_2)         0.14       1.21 r
enable_generate/U228/op (wnor2_2)          0.11       1.32 f
enable_generate/U224/op (wnand3_2)         0.12       1.44 r
enable_generate/ENB_1 (ENB_GEN)            0.00       1.44 r
ENB_1 (out)                                0.00       1.44 r
data arrival time                                     1.44
-------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.13 Timing report for enable generate stage.

4.5.2.3 Timing Report for Value Generate Stage

The following timing report (Figure 4.14) shows the worst path in the enable generate stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the outputs in the value generate unit. This observation corresponds to the structure of this pipeline stage (Figure 4.4). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the value generate stage, the maximum delay is 5.83 ns.

```
Startpoint: md_reg/DOUT_reg[0]
            (rising edge-triggered flip-flop)
Endpoint: MD3[11] (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                                 Incr       Path
```

```
---------------------------------------------------------------------------
md_reg/DOUT_reg[0]/ck (wdp_2)                          0.00       0.00 r
md_reg/DOUT_reg[0]/q (wdp_2)                           0.72       0.72 f
md_reg/DOUT[0] (REG_OP)                                0.00       0.72 f
value_generate/MD_IN[0] (VAL_GEN)                      0.00       0.72 f
value_generate/U449/B[0] (VAL_GEN_DW01_add_14_0)       0.00       0.72 f
value_generate/U449/U131/op (winv_2)                   0.15       0.86 r
value_generate/U449/U90/op (wnand2_2)                  0.15       1.02 f
value_generate/U449/U89/op (wnand2_2)                  0.15       1.16 r
value_generate/U449/U117/op (winv_2)                   0.17       1.33 f
value_generate/U449/U92/op (wnand2_2)                  0.11       1.44 r
value_generate/U449/U91/op (wnand2_2)                  0.20       1.64 f
value_generate/U449/U118/op (winv_2)                   0.16       1.80 r
value_generate/U449/U80/op (wnand2_2)                  0.16       1.96 f
value_generate/U449/U96/op (wnand2_2)                  0.11       2.08 r
value_generate/U449/U94/op (wnand2_2)                  0.21       2.28 f
value_generate/U449/U99/op (wnand2_2)                  0.12       2.40 r
value_generate/U449/U97/op (wnand2_2)                  0.24       2.64 f
value_generate/U449/U82/op (wnor2_2)                   0.19       2.83 r
value_generate/U449/U132/op (wor2_2)                   0.20       3.03 r
value_generate/U449/U100/op (wnand2_2)                 0.17       3.20 f
value_generate/U449/U104/op (wnand2_2)                 0.12       3.32 r
value_generate/U449/U102/op (wnand2_2)                 0.24       3.56 f
value_generate/U449/U84/op (wnor2_2)                   0.19       3.75 r
value_generate/U449/U133/op (wor2_2)                   0.20       3.95 r
value_generate/U449/U105/op (wnand2_2)                 0.17       4.12 f
value_generate/U449/U111/op (wnand2_2)                 0.12       4.23 r
value_generate/U449/U109/op (wnand2_2)                 0.24       4.47 f
value_generate/U449/U86/op (wnor2_2)                   0.19       4.66 r
value_generate/U449/U134/op (wor2_2)                   0.20       4.86 r
value_generate/U449/U114/op (wnand2_2)                 0.17       5.03 f
value_generate/U449/U121/op (winv_2)                   0.12       5.15 r
value_generate/U449/U141/op (wxor2_2)                  0.36       5.52 r
value_generate/U449/SUM[11] (VAL_GEN_DW01_add_14_0)    0.00       5.52 r
value_generate/U560/op (winv_2)                        0.15       5.67 f
value_generate/U533/op (wnor2_2)                       0.17       5.83 r
value_generate/MD3[11] (VAL_GEN)                       0.00       5.83 r
MD3[11] (out)                                          0.00       5.83 r
```

```
data arrival time                                                    5.83
-------------------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.14 Timing report for value generate stage.

4.5.2.4 Timing Report for Partial Product Generate Stage

The following timing report (Figure 4.15) shows the worst path in the enable generate stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the outputs in the partial product generate unit. This observation corresponds to the structure of this pipeline stage (Figure 4.5). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the partial product generate stage, the maximum delay is 2.38 ns.

```
Startpoint: cod_2_reg/DOUT_reg[0]
            (rising edge-triggered flip-flop)
Endpoint: MD_COD2_OUT[12]
          (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                              Incr       Path
-------------------------------------------------------------------------
cod_2_reg/DOUT_reg[0]/ck (wdtp_2)                  0.00       0.00 r
cod_2_reg/DOUT_reg[0]/q (wdtp_2)                   0.65       0.65 f
cod_2_reg/DOUT[0] (REG_N4_1)                       0.00       0.65 f
md_coded_generate/COD_2_IN[0] (MD_COD_GEN)         0.00       0.65 f
md_coded_generate/U2548/op (winv_2)                0.21       0.86 r
md_coded_generate/U1593/op (wnor2_2)               0.13       0.99 f
```

```
md_coded_generate/U2716/op (wnand2_2)              0.25       1.24 r
md_coded_generate/U2717/op (winv_2)                0.13       1.37 f
md_coded_generate/U2718/op (winv_2)                0.40       1.77 r
md_coded_generate/U1927/op (wnor2_2)               0.17       1.94 f
md_coded_generate/U2278/op (wnor2_2)               0.18       2.12 r
md_coded_generate/U2275/op (wnand4_2)              0.26       2.38 f
md_coded_generate/MD_COD2[12] (MD_COD_GEN)         0.00       2.38 f
MD_COD2_OUT[12] (out)                              0.00       2.38 f
data arrival time                                             2.38
--------------------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.15 Timing report for partial product generate stage.

4.5.2.5 Timing Report for First Carry Save Addition Stage

The following timing report (Figure 4.16) shows the worst path in the first carry save addition stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the carry-outs. This observation corresponds to the structure of this pipeline stage (Figure 4.6). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and gives out the maximum delay in the stage. For the first carry save addition stage, the maximum delay is 4.06 ns.

```
Startpoint: sel_reg/DOUT_reg[1]
            (rising edge-triggered flip-flop)
Endpoint: COUT1_1[8] (output port clocked by clk)
Path Group: (none)
Path Type: max
```

```
Point                                          Incr       Path
-------------------------------------------------------------
sel_reg/DOUT_reg[1]/ck (wdp_2)                 0.00       0.00 r
sel_reg/DOUT_reg[1]/q (wdp_2)                  0.50       0.50 f
sel_reg/DOUT[1] (REG_N2)                       0.00       0.50 f
U324/op (winv_2)                               0.19       0.69 r
U261/op (wnand2_2)                             0.15       0.84 f
U345/op (winv_2)                               0.50       1.34 r
U348/op (winv_2)                               0.73       2.07 f
U352/op (wnand2_2)                             0.39       2.46 r
U353/op (winv_2)                               0.25       2.71 f
U354/op (winv_2)                               0.45       3.16 r
U166/op (wnand2_2)                             0.19       3.35 f
U267/op (wnand2_2)                             0.24       3.59 r
csa_11/A[7] (CSA_1_N17_1)                      0.00       3.59 r
csa_11/U216/op (winv_2)                        0.12       3.71 f
csa_11/U146/op (wnand2_2)                      0.11       3.82 r
csa_11/U184/op (wnand2_2)                      0.13       3.95 f
csa_11/U182/op (wnand2_2)                      0.11       4.06 r
csa_11/COUT[8] (CSA_1_N17_1)                   0.00       4.06 r
COUT1_1[8] (out)                               0.00       4.06 r
data arrival time                                         4.06
-------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.16 Timing report for first carry save addition stage.

4.5.2.6 Timing Report for First Carry Propagate Addition Stage

The following timing report (Figure 4.17) shows the worst path in the first carry propagate addition stage. The startpoint of this worst path is one of the register inputs, while the endpoint is one of the intermediate results. This observation corresponds to the structure of this pipeline stage (Figure 4.7). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions

to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the first carry propagate addition stage, the maximum delay is 4.57 ns.

```
Startpoint: cout1_1_reg/DOUT_reg[1]
            (rising edge-triggered flip-flop)
Endpoint: RESULT2_0[15]
          (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                          Incr       Path
-----------------------------------------------------------------
cout1_1_reg/DOUT_reg[1]/ck (wdtp_2)          0.00       0.00 r
cout1_1_reg/DOUT_reg[1]/q (wdtp_2)           0.73       0.73 f
cout1_1_reg/DOUT[1] (REG_N18_3)              0.00       0.73 f
cla_11/B[1] (CLA_1_2)                        0.00       0.73 f
cla_11/U301/op (winv_2)                      0.11       0.84 r
cla_11/U206/op (wnand2_2)                    0.13       0.97 f
cla_11/U254/op (wnand2_2)                    0.19       1.16 r
cla_11/U182/op (wnor2_2)                     0.11       1.27 f
cla_11/U181/op (wnor2_2)                     0.17       1.44 r
cla_11/U193/op (wor2_2)                      0.19       1.63 r
cla_11/U191/op (wnand2_2)                    0.12       1.75 f
cla_11/U215/op (wnand2_2)                    0.11       1.86 r
cla_11/U213/op (wnand2_2)                    0.17       2.04 f
cla_11/U263/op (winv_2)                      0.10       2.13 r
cla_11/U262/op (wnand2_2)                    0.25       2.38 f
cla_11/U227/op (wnand2_2)                    0.13       2.51 r
cla_11/U271/op (wnand2_2)                    0.24       2.76 f
cla_11/U238/op (wnand2_2)                    0.13       2.88 r
cla_11/U237/op (wnand2_2)                    0.20       3.08 f
cla_11/U240/op (wnand2_2)                    0.12       3.20 r
cla_11/U239/op (wnand2_2)                    0.21       3.41 f
cla_11/U243/op (wnand2_2)                    0.12       3.53 r
cla_11/U242/op (wnand2_2)                    0.21       3.75 f
```

```
cla_11/U246/op (wnand2_2)                0.12       3.87 r
cla_11/U245/op (wnand2_2)                0.16       4.03 f
cla_11/U329/op (wxor2_2)                 0.34       4.37 r
cla_11/SUM[15] (CLA_1_2)                 0.00       4.37 r
U69/op (wand2_2)                         0.20       4.57 r
RESULT2_0[15] (out)                      0.00       4.57 r
data arrival time                                   4.57
-------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.17 Timing report for first carry propagate addition stage.

4.5.2.7 Timing Report for Second Carry Save Addition Stage

The following timing report (Figure 4.18) shows the worst path in the second carry save addition stage. The startpoint of this worst path is one of the register inputs, while the endpoint is the partial sum. This observation corresponds to the structure of this pipeline stage (Figure 4.8). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the second carry save addition stage, the maximum delay is 4.11 ns.

```
Startpoint: sel_reg/DOUT_reg[1]
            (rising edge-triggered flip-flop)
Endpoint: RESULT3[20]
          (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                       Incr       Path
```

```
-------------------------------------------------------------
sel_reg/DOUT_reg[1]/ck (wdp_2)              0.00       0.00 r
sel_reg/DOUT_reg[1]/q (wdp_2)               0.87       0.87 r
sel_reg/DOUT[1] (REG_N2)                    0.00       0.87 r
U122/op (winv_2)                            0.33       1.21 f
U141/op (wnand2_2)                          0.37       1.58 r
U142/op (winv_2)                            0.42       2.00 f
U95/op (wnand2_2)                           0.15       2.15 r
U144/op (wnand2_2)                          1.12       3.27 f
csa_2/C[20] (CSA_1_N23)                     0.00       3.27 f
csa_2/U175/op (wxor2_2)                     0.56       3.83 f
csa_2/U152/op (wxor2_2)                     0.29       4.11 f
csa_2/RESULT1[20] (CSA_1_N23)               0.00       4.11 f
RESULT3[20] (out)                           0.00       4.11 f
data arrival time                                      4.11
-------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.18 Timing report for second carry save addition stage.

4.5.2.8 Timing Report for Second Carry Propagate Addition Stage

The following timing report (Figure 4.19) shows the worst path in the second carry propagate addition stage. The startpoint of this worst path is one of the register inputs, while the endpoint is the final result. This observation corresponds to the structure of this pipeline stage (Figure 4.9). The first column of the timing table indicates the logic gates or wires on this worst path, the second column shows their contributions to the total delay, and the third column accumulates all these delays and provides the maximum delay in the stage. For the second carry propagate addition stage, the maximum delay is 4.91 ns.

```
Startpoint: result3_reg/DOUT_reg[16]
            (rising edge-triggered flip-flop)
Endpoint: RESULT_FINAL[22]
          (output port clocked by clk)
Path Group: (none)
Path Type: max

Point                                                  Incr       Path
-----------------------------------------------------------------------
result3_reg/DOUT_reg[16]/ck (wdtp_2)                   0.00       0.00 r
result3_reg/DOUT_reg[16]/q (wdtp_2)                    0.67       0.67 f
result3_reg/DOUT[16] (REG_N23_1)                       0.00       0.67 f
fa_1/A[0] (FANBIT_N7_1)                                0.00       0.67 f
fa_1/U66/op (winv_2)                                   0.13       0.80 r
fa_1/U29/op (wnand2_2)                                 0.13       0.93 f
fa_1/U38/op (wnand2_2)                                 0.11       1.04 r
fa_1/U36/op (wnand2_2)                                 0.18       1.21 f
fa_1/U54/op (winv_2)                                   0.16       1.37 r
fa_1/U31/op (wnand2_2)                                 0.16       1.53 f
fa_1/U41/op (wnand2_2)                                 0.11       1.64 r
fa_1/U39/op (wnand2_2)                                 0.24       1.88 f
fa_1/U32/op (wnor2_2)                                  0.19       2.07 r
fa_1/U44/op (wor2_2)                                   0.19       2.27 r
fa_1/U42/op (wnand2_2)                                 0.17       2.44 f
fa_1/U55/op (winv_2)                                   0.16       2.59 r
fa_1/U33/op (wnand2_2)                                 0.16       2.75 f
fa_1/U47/op (wnand2_2)                                 0.11       2.86 r
fa_1/U45/op (wnand2_2)                                 0.24       3.11 f
fa_1/U34/op (wnor2_2)                                  0.19       3.30 r
fa_1/U50/op (wor2_2)                                   0.19       3.49 r
fa_1/U48/op (wnand2_2)                                 0.17       3.66 f
fa_1/U56/op (winv_2)                                   0.16       3.82 r
fa_1/U35/op (wnand2_2)                                 0.16       3.98 f
fa_1/U53/op (wnand2_2)                                 0.11       4.09 r
fa_1/U51/op (wnand2_2)                                 0.24       4.33 f
fa_1/U67/op (wxor2_2)                                  0.35       4.68 r
fa_1/S[6] (FANBIT_N7_1)                                0.00       4.68 r
U12/op (wmux2_2)                                       0.23       4.91 r
```

```
RESULT_FINAL[22] (out)                          0.00       4.91 r
data arrival time                                          4.91
-------------------------------------------------------------------------
(Path is unconstrained)
```

Figure 4.19 Timing report for second carry propagate addition stage.

4.5.2.9 Summary of Maximum Delay for Each Pipeline Stage

According to the timing reports, the maximum delay of each pipeline stage is summarized in Table 4.3 (Total latency: 29.57 ns). Two observations can be made. First, the maximum delay of all pipeline stages occurs in the value generate stage, which corresponds to the fact that the value of 3*multiplicand is calculated in this stage. This calculation involves the major power-consuming operation of multipliers – addition – as indicated in Chapter3, and the addition carried out here is not manually optimized. Second, the value of this maximum delay is 5.83ns; thus, the throughput of the core can be estimated using Equation 4.1 as $1/(5.83 \times 10^{-9}) \approx 171.5$ MIPS.

Pipeline Stage	Maximum Delay (ns)
Code Generate Stage	2.27
Enable Generate Stage	1.44
Value Generate Stage	5.83
Partial Product Generate Stage	2.38
First Carry Save Addition Stage	4.06
First Carry Propagate Addition Stage	4.57
Second Carry Save Addition Stage	4.11
Second Carry Propagate Addition Stage	4.91

Table 4.3 Maximum delay of each pipeline stage.

4.5.3 Power

As indicated in Chapter 3, the major power-consuming operation for a multiplier is addition. Various recoding schemes can effectively reduce the number of additions performed during multiplication, and thus decrease the power consumption. One feature of this reusable multiplication component is its power-efficiency compared to that of conventional multipliers. Moreover, this feature was demonstrated through simulation results showing a large reduction in the number of additions required for multiplication.

The modeling file and testbench were modified in order to compare the number of additions performed during multiplication using recoding schemes to that without recoding. Simulation was then run using 500 random test vectors, and the result is summarized in Table 4.4.

Mode	More Addition	Equal Addition	Less Addition
8-bit operands, radix-4 recoding	7.8%	25.4%	66.8%
8-bit operands, radix-8 recoding	23.8%	23.6%	52.6%
12-bit operands, radix-4 recoding	6%	12%	82%
12-bit operands, radix-8 recoding	10%	2%	88%

Table 4.4 Simulation result comparing addition needed with and without recoding.

The table shows that, regardless of the recoding scheme and operand wordlength, the use of recoding schemes can have at least a 52.6% chance of reducing the number of additions needed for multiplication, and thus effectively lower power-consumption and increase power-efficiency.

4.6 Design for Testability of the Multiplication Component

Design verification for this reusable multiplication component core is function oriented, that is, to verify that the system meets functionality requirements. The exhaustive approach for verification may also be used for module testing. However, the resulting test pattern is too long to be practical since a long test pattern leads to an excessively high testing cost. In order to keep testing costs within reasonable bounds while still achieving high fault coverage, design-for-testability (DFT) techniques are highly recommended for contemporary chip designs. Generally speaking, controllability and observability are the two most important factors in determining the testability of a device [81].

As an extension of this thesis research, several DFT techniques are suggested in this section in order to achieve better observability and controllability for the testing of this multiplication component. These include scannable registers, the AND/OR tree based technique [82], and the C-testable technique for iterative logic array [81].

4.6.1 Scannable Registers

An effective way of improving the observability and controllability of the internal states in a digital system is to make the registers in the system directly accessible during the test. For this multiplication component, doing so means the tester can arbitrarily control and easily observe the bit value in each register. This is achieved through the use of the suggested scanable register (SRs). These registers have four modes: normal, shift, snapshot and test mode (Figure 4.20). In the normal/shift/snapshot/test mode, data are applied from normal/test/normal/test data input ports and come out at normal/test/test/normal data output ports.

Mode Select	Mode
00	Normal mode
01	Shift mode
10	Snapshot mode
11	Test mode

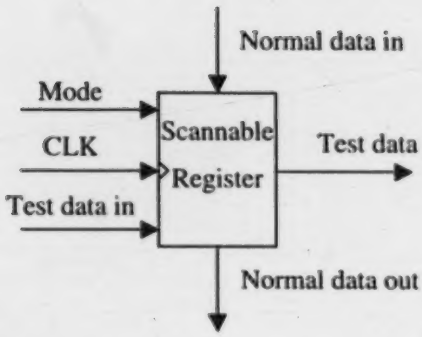


Figure 4.20 Scannable registers.

4.6.2 AND/OR Tree Based Technique

The AND/OR tree based technique was originally used in the testing of configurable logic blocks (CLBs) in field programmable gate arrays (FPGAs) [82]. If circuits under test (CUTs) all have the same architecture, then outputs of these CUTs must be the same under the same test pattern if they are fault-free.

Figure 4.21 shows the basic structure for three CUTs and under the condition that one output of each CUT is considered at a time. Given an AND tree made up of 2-input AND gates, if each leaf of the tree is taken as an input (I_n), the output is given by OUT = $I_1 \cap I_2 \cap \ldots \cap I_n$. Similarly, for an OR tree, OUT = $I_1 \cup I_2 \cup \ldots \cup I_n$. The AND/OR tree has the property that the effect of a stuck-at-0/1 fault on any input, or any other faults that make the logic value of the input change from 1(0) to 0(1), can always be propagated to the output (root) of the AND/OR tree if all other inputs of the tree are at logic value 1(0). The AND/OR tree technique can be applied to this multiplication component because it contains identical sub-components such as adders and flip-flops. The feature of this technique is that it has strong multi-fault detectability. With an exhaustive test for each CUT, the AND/OR tree can completely detect any practical multiple-fault CUTs.

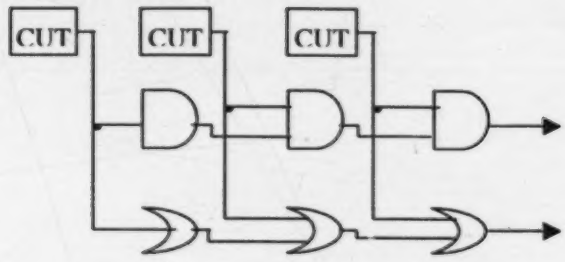


Figure 4.21 ADD/OR tree based structure.

4.6.3 C-testable Technique for Iterative Logic Arrays (ILAs)

Pseudo-exhaustive testing techniques based on partitioning are perfectly suited for circuit structures such as ILAs, which are composed of identical CUTs interconnected in a regular pattern. Figure 4.22 shows an ILA with three CUTs. The partitioning problem is solved by exhaustively testing every CUT. Combined with the SRs shown in Figure 4.20, the ILAs have the useful property that they can be pseudo-exhaustively tested with a number of tests that depend not on the number of CUTs in the ILA but only on one CUT, which is said to be "constant testable" (C-testable).

In the normal operation mode, the data (c_i) goes into each SR from "normal data in" and comes out (c_i') at "normal data out." When the circuit is under test, the applied test patterns (s_i) are first shifted into the SRs under "shift mode," and then applied to each CUT (c_i') under "test mode."

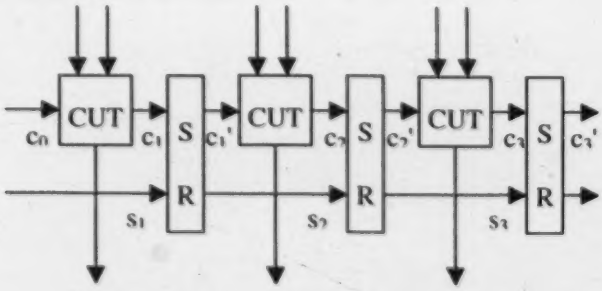


Figure 4.22 C-testable structure for an ILA.

4.6.4 Application to the Multiplication Component

The developed multiplication component can be re-divided into two main parts: the recoding circuit and the addition circuit, with the latter incorporating the final two custom addition sections shown in Figure 4.1.

4.6.4.1 AND/OR Tree for the Recoding Circuit

The recoding circuit produces the partial products following either the radix-4 or 8 Booth recoding scheme and depending on slices of the multiplier. Figure 4.23(a) shows how to generate the partial products applicable to the AND/OR tree.

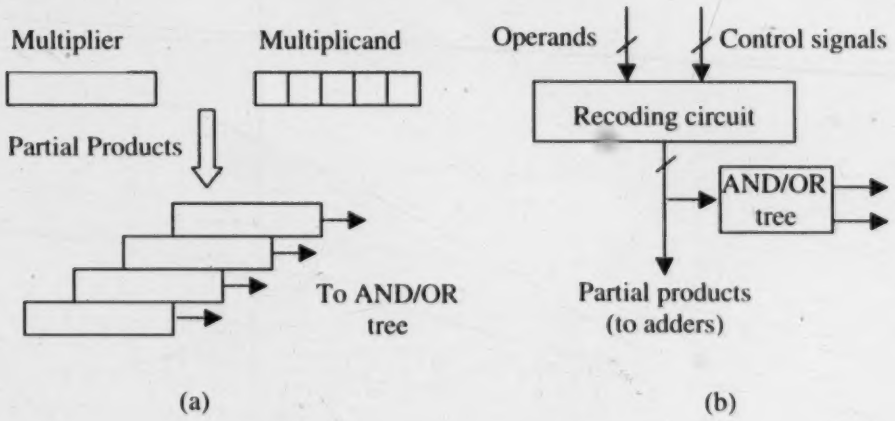


Figure 4.23 Coding circuit with AND/OR trees.

The radix-4 Booth recoding scheme can be used as an example. It generates a partial product after examining three bits of the multiplier at a time. One partial product may not be the same as the next generated partial product since they share a common bit but at a different location. However, this problem can be solved when they are under test (not in

the normal system operation mode) by replacing the common bit of slices with a bit value from either of the two independent registers storing one or zero. Therefore, each partial product can be the same as other partial-products and hence be exhaustively generated and tested. Figure 4.23(b) shows the recoding circuit with the AND/OR tree.

4.6.4.2 C-Testable Technique for the Adders

The addition circuit can be divided into smaller sub-units with the same function. Figure 4.24 shows the application of C-testable technique on a 16-bit carry lookahead adder.

Using the C-testable technique, the 4-bit subadders and the carry generation circuit can be exhaustively tested with the number of test vectors being greatly reduced (from $2^{16}\times2^{16}$ to $4\times2^{4}\times2^{4}$).

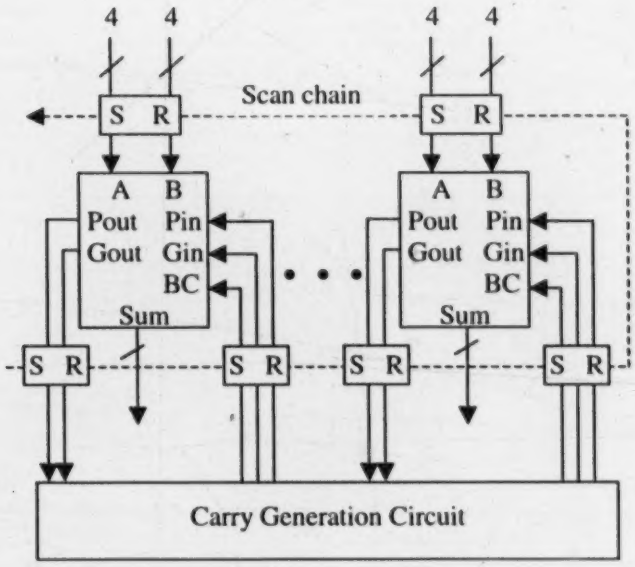


Figure 4.24 Application of C-testable technique to a CLA adder.

Chapter 5

Conclusions and Recommendations

Digital signal processing systems have rapidly increased in demand due to the prevalence of various wireless communication devices. However, their application is still restricted by adaptability, throughput and power-consumption limitations. Research into baseband DSP systems with better adaptabiltiy, higher throughput, and more power efficiency has been carried out at algorithmic and architectural levels. On the other hand, superior overall systems can also be achieved through the research and development of various baseband DSP components with better performance in terms of adaptability, throughput, and power efficiency. This applies especially to multiplication components, largely due to their importance in the performance of the whole baseband DSP system.

During this thesis research, comprehensive reviews on (i) low-power wireless multimedia communication systems and (ii) addition and multiplication schemes for baseband digital signal processing components were first carried out. A multiplication component was then developed based on the review and understanding of underlying principles and past work. The development of this component also reflects a design concept that could be applied to the development of reusable embedded component cores for SoC designs. This concept consists of three areas of focus, namely, functionality, structure, and design methodology. The development process and results prove that the design concept could be adopted in order to achieve retargetability, high throughput, and reduced power consumption.

Retargetability is one of the functionality requirements. It is demonstrated through the core's ability to switch between radix-4 and 8 recoding schemes for carrying out multiplication, and its capability to handle different operand lengths: 8- and 12-bit two's complement operands, with the two operands accepted simultaneously being of the same length. The calculation results are also in two's complement form. The choice for different radix recoding schemes and operand widths is sent to the core through two control signal bits.

Throughput of this multiplication component core is expected to exceed 150 MIPS and is achieved through the incorporation of architectural level design styles like pipelining and parallelism. The whole core is divided into eight pipeline stages, while parallelism is utilized within several pipeline stages. Besides these, various high performance addition and multiplication schemes like carry save, carry lookahead, carry select, and Booth recoding approaches are adopted so as to speed up calculation.

Low power operation compared to that of conventional multipliers, i.e., array multipliers, is expected to be obtained. This result is achieved by using Booth recoding schemes, which reduce the most power-consuming operation in all multipliers – addition. The usage of the carry save addition scheme also contributes to the overall power efficiency.

After the research and development, this multiplication component core underwent extensive verification and simulation. This process was necessary due to the lack of previous results and reporting. Exhaustive testing showed that the prescribed retargetability was successfully achieved. Also, the estimated throughput after synthesis is 171.5 MIPS, which exceeds the expected 150 MIPS. As well, this component needs at least 52.6% fewer addition operations than a conventional multiplier, and thus effectively reduces power consumption and meets the expected objective.

Besides the above-mentioned functional features, a formal methodology was adopted during the development of this multiplication component core as part of the design concept, in order to make the core reusable and ready for integration into the System-on-a-Chip (SoC) designs that are being increasingly accepted and supported in the semiconductor industry.

Finally, as an extension of this thesis research, it is recommended that design-for-testability considerations be incorporated into this multiplication component core. Several suitable approaches have been proposed and can be incorporated into the core in the future.

Bibliography

[1] A. Gatherer, et al., "DSP-based architectures for mobile communications: past, present and future," in IEEE Commun. Mag., Vol. 38, No. 1, 2000, pp. 84 – 90.

[2] K. Feher, Wireless Digital Communications, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[3] L.E. Larson, "Integrated circuit technology options for RFIC's – present status and future directions," in IEEE Journal of Solid-State Circuits, Vol. 33, No. 3, 1998, pp. 387 – 399.

[4] B. Razavi, RF Microelectronics, Prentice-Hall, Upper Saddle River, NJ, 1998.

[5] J.G. Proakis and D.G. Manolakis, Digital Signal Processing: Principles, Algorithms, and Applications, 3rd ed., Prentice-Hall, Upper Saddle River, NJ, 1996.

[6] J.S. Eager, "Advances in rechargeable batteries spark product innovation," in Proc. 1992 Silicon Valley Computer Conf., 1992, pp. 243 – 253.

[7] A.S. Tanenbaum, Computer Networks, 3rd ed., Prentice-Hall, Upper Saddle River, NJ, 1996.

[8] P. Agrawal, "Energy conservation design techniques for mobile wireless VLSI systems," in Proc. IEEE Computer Society Workshop on VLSI'98, 1998, pp. 34 – 39.

[9] J.-C. Chen, K.M. Sivalingam, P. Agrawal, and S. Kishore, "A comparison of MAC protocols for wireless local networks based on battery power consumption," in Proc. IEEE INFOCOM, 1998, pp. 150 – 157.

[10] IEEE, "Wireless LAN medium access control (MAC) and physical layer (PHY) Spec," P802.11/D5, Draft Standard IEEE 802.11, May 1996.

[11] D.J. Goodman, R.A. Valenzuela, K.T. Gayliard, and B. Ramamurthi, "Packet reservation multiple access for local wireless communications," IEEE Transactions on Communications, Vol. 37, 1989, pp. 885 – 890.

[12] D. Raychaudhuri and N.D. Wilson, "ATM-based transport architecture for multi-services wireless personal communication networks," IEEE Journal on Selected Areas in Communications, Vol. 12, 1994, pp. 1401 – 1414.

[13] K.M. Sivalingam, M.B. Srivastava, P. Agrawal, and J.-C. Chen, "Low-power access protocols based on scheduling for wireless and mobile ATM networks," in Proc. IEEE International Conference on Universal Personal Communications, 1997, pp. 429 – 433.

[14] M.J. Karol, Z. Liu, and K.Y. Eng, "An efficient demand-assignment multiple access protocol for wireless packet (ATM) networks," in ACM/Baltzer Wireless Networks, Vol. 1, 1995, pp. 267 – 279.

[15] T.E. Truman, R.W. Brodersen, "A design methodology for highly-integrated wireless communications systems," in Proc. IEEE Computer Society Workshop on VLSI'98, 1998, pp. 66 – 70.

[16] S. Narayanaswamy, et al., "Application and network support for InfoPad," in IEEE Personal Communications, April 1996, pp. 4 – 17.

[17] R. Want, et al., "An overview of the ParcTab ubiquitous computing experiment," in IEEE Personal Communications, December 1995, pp. 28 – 43.

[18] A.A. Abidi, "Low-power radio-frequency IC's for portable communications," in Proceedings of the IEEE, Vol. 83, No. 4, 1995, pp. 544 – 569.

[19] P.R. Gray and R.G. Meyer, "Future directions in silicon ICs for RF personal Communications," in Proc. of the Custom Integrated Circuits Conf., 1995, pp. 83 – 90.

[20] W.G. Kasperkovitz, "An integrated FM receiver," in Microelectronics Reliability, Vol. 21, No. 2, 1981, pp. 183 – 189.

[21] T. Okanobu, H. Tomiyama, and H. Arimoto, "Advanced low-voltage single chip radio IC," in IEEE Trans. on Consumer Electronics, Vol. 38, No. 3, 1992, pp. 465 – 475.

[22] I.A.W. Vance, "Fully integrated radio paging receiver," in IEE Proc. Part F, Vol. 129, No. 1, 1982, pp. 2 – 6.

[23] J. Crols and M. Steyaert, "A single-chip 900 MHz CMOS receiver front-end with a high performance low-IF topology," in IEEE Journal of Solid-State Circuits, Vol. 30, 1995, pp. 1483 – 1492.

[24] A.P. Chandrakasan, S. Sheng and R.W. Brodersen, "Low-power CMOS digital design," in IEEE Journal of Solid-State Circuits, Vol. 27, No. 4, 1992, pp. 473 – 484.

[25] N.H.E. Weste and K. Eshraghian, Principles of CMOS VLSI Design: A Systems Perspective, 2nd ed., Addison-Wesley, Reading, MA, 1993.

[26] D.A. Patterson and J.L. Hennessy, Computer Architecture: A Quantitative Approach, 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[27] C.E. Leiserson, F. Rose and J. Saxe, "Optimizing synchronous circuitry by retiming," in Proc. of the Third Caltech Conference on VLSI, March 1983, pp. 87 – 116.

[28] K.K. Parhi and D.G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," in IEEE Trans. on Computers, Vol. 40, 1991, pp. 178 – 195.

[29] K.K. Parhi, "A systematic approach for design of digit-serial signal processing architectures," in IEEE Trans. on Circuits and Systems, Vol. 38, 1991, pp. 358 – 375.

[30] C. Wang and K.K. Parhi, "High-level DSP synthesis using concurrent transformation, scheduling, and allocation," in IEEE Trans. CAD, Vol. 14, No. 3, 1995, pp. 274 – 295.

[31] K.K. Parhi and D.G. Messerschmitt, "Pipeline interleaving and parallelism in recursive idgital filters – part I: pipelining using scattered look-ahead and decomposition," in IEEE Trans. Acoustic, Speech, Signal Processing, Vol. 37, No. 7, 1989, 1099 – 1117.

[32] S.-H. Huang and J.M. Rabaey, "Maximizing the throughput of high performance DSP applications using behavioral transformations," Proc. EDAC-EUROASIC, March 1994, pp. 25 – 30.

[33] T.Arslan, A.T. Erdogan and D.H. Horrocks, "Low power design for DSP: methodologies and techniques," in Microelectronics Journal, Vol. 27, 1996, pp. 731 – 744.

[34] K.K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis," in Journal of VLSI Signal Processing, Vol. 9, 1995, pp. 121 – 143.

[35] A. Bellaouar and M.I. Elmasry, Low-power Digital VLSI Design: Circuits and Systems, Kluwer Academic Publishers, Boston, MA, 1995.

[36] D.B. Lidsky and J.M. Rabaey, "Low-power design of memory intensive functions," in IEEE Sym. on Low Power Electronics, Tech. Dig., October 1994, pp. 16 – 17.

[37] A. Chandrakasan and R. Brodersen (eds), Low-power CMOS Design, IEEE Press, New York, NY, 1998.

[38] G.K. Yeap, Practical Low Power Digital VLSI Design, Kluwer Academic Publishers, Boston, MA, 1998.

[39] R.H. Katz, Contemporary Logic Design, Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994.

[40] A.R. Omondi, Computer Arithmetic Systems: Algorithms, Architecture and Implementation, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[41] J.M. Rabaey and M. Pedram (eds), Low Power Design Methodologies, Kluwer Academic Publishers, Boston, MA, 1996.

[42] B. Gilchrist, et al., "Fast carry logic for digital computers," in IRE Trans. EC-4, Dec. 1955, pp. 133 – 136.

[43] S. Turrini, "Optimal group distribution in carry-skip adders," in Proc. of the 9th Sym. on Computer Arithmetic, 1989, pp. 96 – 103.

[44] R.P. Brent and H.T. Kung, "A regular layout for parallel adders," in IEEE Trans. Comput., Vol. C-31, No. 3, 1982, pp. 260 – 264.

[45] O.L. MacSorley, "High-speed arithmetic in binary computers," in IRE Proceedings, Vol. 49, 1961, pp. 67 – 91.

[46] J. Sklansky, "Conditional-sum addition logic," IRE Transactions on Electronic Computers, Vol. EC-9, 1960, pp. 226 – 231.

[47] O.J. Bedrij, "Carry-select adder," IRE Transactions on Electronic Computers, Vol. EC-11, 1962, pp. 340 – 346.

[48] S. Waser and M.J. Flynn, Introduction to Arithmetic for Digital Systems Designers, CBS College Publishing, Taipei, Taiwan, 1983.

[49] K.K. Parhi, "A systematic approach for design of digit-serial signal processing architectures," in IEEE Trans. on Circuits and Systems, Vol. 38, No. 4, 1991, pp. 358 – 375.

[50] H.R. Srinivas and K.K. Parhi, "A fast VLSI adder architecture," in IEEE JSSC, Vol. 27, No. 5, 1992, pp. 761 – 767.

[51] J.M. Dobson and G.M. Blair, "Fast two's complement VLSI adder design," in Electronics Letters, Vol. 31, No. 20, Sept. 1995, pp. 1721 – 1722.

[52] H. Makino, et al., "An 8.8-ns 54x54-bit multiplier with high speed redundant binary architecture," in IEEE JSSC, Vol. 31, No. 6, 1996, pp. 773 – 783.

[53] K.K. Parhi, "Fast low-energy VLSI binary addition," in Proc. 1997 IEEE Int'l Conf. on Computer Design (ICCD), 1997, pp. 676 – 684.

[54] K. Hwang, Computer Arithmetic: Principles, Architecture, and Design, John Wiley & Sons, New York, NY, 1979.

[55] I.S. Abu-Khater, A. Bellaouar, and M.I. Elmasry, "Circuit techniques for CMOS low-power high-performance multipliers," in IEEE JSSC, Vol. 31, No. 10, 1996, pp. 1535 – 1546.

[56] C.J. Nicol and P. Larsson, "Low power multiplication for FIR filters," in ISLPED, 1997, pp. 76 – 79.

[57] Y. Hagihara, et al., "A 2.7ns 0.25μm CMOS 54x54b multiplier," in IEEE ISSCC, 1998, pp. 296 – 297.

[58] E.M. Schwarz, R.M. Averill III and L.J. Sigal, "A radix-8 CMOS S/390 multiplier," in Proc. IEEE 13th Sym. on Computer Arithmetic, pp. 2 – 9.

[59] R. Fried, "Minimizing energy dissipation in high-speed multipliers," in ISLPED, 1997, pp. 214 – 219.

[60] C. Lemonds, et al., "A low power 16 by 16 multiplier using transition reduction circuitry," in Proc. of 1994 Int'l Workshop on Low Power Design, 1994, p 139.

[61] S.S. Mahant-Shetti, C. Lemonds and P. Balsara, "Leap frog multiplier," in ISLPED, 1996, pp. 221 – 223.

[62] K.Z. Pekmestzi, "Multiplexer-based array multipliers," in IEEE Trans. on Computers, Vol. 48, No. 1, 1999, pp. 15 – 23.

[63] C. Wallace, "A suggestion for a fast multiplier," in IEEE Trans. On Electronic Computers, Vol. EC-13, Feb. 1964, pp. 14 –17.

[64] D. Carlson, et al., "A 667MHz RISC microprocessor containing a 6.0ns 64b integer multiplier," in IEEE ISSCC, 1998, pp. 294 – 295.

[65] P.C.H. Meier, R.A. Rutenbar and L.R. Carley, "Exploring multiplier architecture and layout for low power," in IEEE 1996 CICC, 1996, pp. 513 – 516.

[66] L. Dadda, "Some schemes for parallel multipliers," in Alta Frequenza, 34, 1965, pp. 349 – 356.

[67] Y.N. Chang, J.H. Satyanarayana and K.K. Parhi, "Low-power digit-serial multipliers," in 1997 IEEE Int'l Sym. on Circuits and Systems, 1997, pp. 2164 – 2167.

[68] Y.N. Chang, J.H. Satyanarayana and K.K. Parhi, "Systematic design of high-speed and low-power digit-serial multipliers," in IEEE Trans. on Circuits and Systems, II: Analog and Digital Signal Processing, Vol. 45, No. 12, 1998, pp. 1585 – 1596.

[69] S.D. Pezaris, "A 40ns 17-bit-by-bit array multiplier," in IEEE Trans. Computers, Vol. C-20, No. 4, Apr. 1971, pp. 442 – 447.

[70] C.R. Baugh and B.A. Wooley, "A two's complement parallel array multiplication algorithm," in IEEE Trans. Computers, Vol. C-22, No. 12, 1973, pp. 1045 – 1047.

[71] A. Hemel, "Making small ROMs do math quickly, cheaply and easily," in Electronic Computer Memory Technology, W.B. Riley, ed., McGraw-Hill, New York, NY, 1971, pp. 133 – 140.

[72] T.A. Brubaker and J.C. becker, "Multiplication using logarithms implemented with read-only memory," in IEEE Trans. Computers, Vol. C-24, 1975.

[73] E. Abu-Shama, M.B. Maaz and M.A. Bayoumi, "A fast and low power multiplier architecture," in Midwest Sym. on Circuits and Systems, Vol. 1, 1996, pp. 53 – 56.

[74] U. Ko, P.T. Balsara, and W. Lee, "Low-power design techniques for high-performance CMOS adders," in IEEE Trans. VLSI Systems, Vol. 3, No. 2, 1995, pp. 327 – 333.

[75] R. Zimmermann and R Gupta, "Low-power logic styles," in ESSCIRC, 1996.

[76] B. Ackland and C Nicol, "High performance DSPs – what's hot and what's not?" in ISLPED, 1998, pp. 1 – 6.

[77] M. Suzuki, et al., "A 1.5-ns 32-b CMOS ALU in double pass-transistor logic," in IEEE JSSC, Vol. 28, No. 11, 1993, pp. 1145 – 1151.

[78] M. Margala and N.G. Durdle, "Low-voltage power-efficient BiDPL adder for VLSI applications," in Microelectronics Journal, Vol. 30, No. 2, 1999, pp.193-197.

[79] A.M. Shams and M.A. Bayoumi, "A structured approach for designing low power adders," in Conf. Record of the Asilomer Conf. on Signals, Systems, and Computers, Vol. 1, 1997, pp. 757 – 761.

[80] A. Sayed and M. Bayoumi, "A new low power building block cell for adders," in Proc. 1997 40th Midwest Sym. on Circuits and Systems, Part 2 of 2, 1997, pp. 818 – 822.

[81] M. Abramovici, M.A. Breuer and A.D. Friedman, Digital System Testing and Testable Design, IEEE Computer Society Press, Piscataway, NJ, 1992.

[82] W.K. Huang, F.J. Meyer and F. Lombardi, "Multiple fault detection in logic resources of FPGAs," in Proc. IEEE Int'l Workshop on Defect and Fault Tolerance in VLSI Systems, 1997, pp. 186 – 194.

[83] J. Crols and M. Steyaert, CMOS Wireless Transceiver Design, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

Appendices

Appendix 1: Structure of Code Generate Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Code Generate Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity REC_1 is
  port (
    MD_IN, MR_IN              : in  std_logic_vector(11 downto 0);
    CLK, RST                  : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN    : in  std_logic;
    MD_OUT                    : out std_logic_vector(11 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT : out std_logic;
    COD_5, COD_4, COD_3       : out std_logic_vector( 3 downto 0);
    COD_2, COD_1, COD_0       : out std_logic_vector( 3 downto 0));
end REC_1;

architecture STR of REC_1 is

  component REG_OP
    port (
      DIN           : in  std_logic_vector(11 downto 0);
      CLK, RST, ENB : in  std_logic;
      DOUT          : out std_logic_vector(11 downto 0));
  end component;
```

```
  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component COD_GEN
    port (
      MR_IN                   : in  std_logic_vector(11 downto 0);
      SEL_RAD_IN, SEL_BIT_IN  : in  std_logic;
      COD_5, COD_4, COD_3     : out std_logic_vector( 3 downto 0);
      COD_2, COD_1, COD_0     : out std_logic_vector( 3 downto 0));
  end component;

  signal MR                : std_logic_vector(11 downto 0);
  signal SEL_IN, SEL_OUT   : std_logic_vector( 1 downto 0);

begin

  SEL_IN <= SEL_RAD_IN & SEL_BIT_IN;

  md_reg : REG_OP
    port map (
      DIN  => MD_IN,
      CLK  => CLK,
      RST  => RST,
      ENB  => SEL_BIT_IN,
      DOUT => MD_OUT);

  mr_reg : REG_OP
    port map (
      DIN  => MR_IN,
      CLK  => CLK,
      RST  => RST,
      ENB  => SEL_BIT_IN,
      DOUT => MR);

  sel_reg : REG
    generic map (
      N => 2)
    port map (
      DIN  => SEL_IN,
      CLK  => CLK,
      RST  => RST,
      DOUT => SEL_OUT);

  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);

  code_generate : COD_GEN
    port map (
      MR_IN      => MR,
      SEL_RAD_IN => SEL_OUT(1),
```

```
      SEL_BIT_IN => SEL_OUT(0),
      COD_5      => COD_5,
      COD_4      => COD_4,
      COD_3      => COD_3,
      COD_2      => COD_2,
      COD_1      => COD_1,
      COD_0      => COD_0);

end STR;
```

Appendix 2: Structure of Enable Generate Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Enable Generate Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity REC_2 is
  port (
    MD_IN                          : in  std_logic_vector(11 downto 0);
    CLK, RST                       : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN         : in  std_logic;
    COD_5_IN, COD_4_IN, COD_3_IN   : in  std_logic_vector( 3 downto 0);
    COD_2_IN, COD_1_IN, COD_0_IN   : in  std_logic_vector( 3 downto 0);
    MD_OUT                         : out std_logic_vector(11 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT       : out std_logic;
    COD_5_OUT, COD_4_OUT, COD_3_OUT: out std_logic_vector( 3 downto 0);
    COD_2_OUT, COD_1_OUT, COD_0_OUT: out std_logic_vector( 3 downto 0);
    ENB_N4, ENB_4, ENB_N3, ENB_3   : out std_logic;
    ENB_N2, ENB_2, ENB_N1, ENB_1   : out std_logic);
end REC_2;

architecture STR of REC_2 is

  component REG_OP
    port (
      DIN           : in  std_logic_vector(11 downto 0);
      CLK, RST, ENB : in  std_logic;
      DOUT          : out std_logic_vector(11 downto 0));
  end component;

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
```

```
    DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component ENB_GEN
    port (
      COD_5, COD_4, COD_3          : in  std_logic_vector(3 downto 0);
      COD_2, COD_1, COD_0          : in  std_logic_vector(3 downto 0);
      ENB_N4, ENB_4, ENB_N3, ENB_3 : out std_logic;
      ENB_N2, ENB_2, ENB_N1, ENB_1 : out std_logic);
  end component;

  signal COD_5, COD_4, COD_3 : std_logic_vector(3 downto 0);
  signal COD_2, COD_1, COD_0 : std_logic_vector(3 downto 0);
  signal SEL_IN, SEL_OUT     : std_logic_vector(1 downto 0);

begin

  SEL_IN      <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);
  COD_5_OUT   <= COD_5;
  COD_4_OUT   <= COD_4;
  COD_3_OUT   <= COD_3;
  COD_2_OUT   <= COD_2;
  COD_1_OUT   <= COD_1;
  COD_0_OUT   <= COD_0;

  md_reg : REG_OP
    port map (
      DIN => MD_IN, CLK => CLK, RST => RST, ENB => SEL_BIT_IN,
      DOUT => MD_OUT);

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  cod_5_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_5_IN, CLK => CLK, RST => RST, DOUT => COD_5);

  cod_4_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_4_IN, CLK => CLK, RST => RST, DOUT => COD_4);

  cod_3_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_3_IN, CLK => CLK, RST => RST, DOUT => COD_3);

  cod_2_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_2_IN, CLK => CLK, RST => RST, DOUT => COD_2);
```

```
  cod_1_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_1_IN, CLK => CLK, RST => RST, DOUT => COD_1);

  cod_0_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_0_IN, CLK => CLK, RST => RST, DOUT => COD_0);

  enable_generate : ENB_GEN
    port map (
      COD_5 => COD_5, COD_4 => COD_4, COD_3 => COD_3,
      COD_2 => COD_2, COD_1 => COD_1, COD_0 => COD_0,
      ENB_N4 => ENB_N4, ENB_4 => ENB_4, ENB_N3 => ENB_N3,
      ENB_3 => ENB_3, ENB_N2 => ENB_N2, ENB_2 => ENB_2,
      ENB_N1 => ENB_N1, ENB_1 => ENB_1);

end STR;
```

Appendix 3: Structure of Value Generate Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Value Generate Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity REC_3 is
  port (
    MD_IN                          : in  std_logic_vector(11 downto 0);
    CLK, RST                       : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN         : in  std_logic;
    COD_5_IN, COD_4_IN, COD_3_IN   : in  std_logic_vector( 3 downto 0);
    COD_2_IN, COD_1_IN, COD_0_IN   : in  std_logic_vector( 3 downto 0);
    ENB_N4, ENB_4, ENB_N3, ENB_3   : in  std_logic;
    ENB_N2, ENB_2, ENB_N1, ENB_1   : in  std_logic;
    SEL_RAD_OUT, SEL_BIT_OUT       : out std_logic;
    COD_5_OUT, COD_4_OUT, COD_3_OUT: out std_logic_vector( 3 downto 0);
    COD_2_OUT, COD_1_OUT, COD_0_OUT: out std_logic_vector( 3 downto 0);
    N_MD4, MD4, N_MD3, MD3         : out std_logic_vector(13 downto 0);
    N_MD2, MD2, N_MD1, MD1        : out std_logic_vector(13 downto 0));
end REC_3;

architecture STR of REC_3 is

  component REG_OP
    port (
```

```
    DIN           : in  std_logic_vector(11 downto 0);
    CLK, RST, ENB : in  std_logic;
    DOUT          : out std_logic_vector(11 downto 0));
  end component;

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component VAL_GEN
    port (
      MD_IN                          : in  std_logic_vector(11 downto 0);
      SEL_RAD_IN, SEL_BIT_IN         : in  std_logic;
      ENB_N4, ENB_4, ENB_N3, ENB_3 : in  std_logic;
      ENB_N2, ENB_2, ENB_N1, ENB_1 : in  std_logic;
      N_MD4, MD4, N_MD3, MD3         : out std_logic_vector(13 downto 0);
      N_MD2, MD2, N_MD1, MD1       : out std_logic_vector(13 downto 0));
  end component;

  signal MD              : std_logic_vector(11 downto 0);
  signal SEL_IN, SEL_OUT : std_logic_vector( 1 downto 0);
  signal ENB_IN, ENB_OUT : std_logic_vector( 7 downto 0);

begin

  SEL_IN      <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);
  ENB_IN      <= ENB_N4 & ENB_4 & ENB_N3 & ENB_3 & ENB_N2 & ENB_2 &
ENB_N1 & ENB_1;

  md_reg : REG_OP
    port map (
      DIN => MD_IN, CLK => CLK, RST => RST, ENB => SEL_BIT_IN,
      DOUT => MD);

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  cod_5_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_5_IN, CLK => CLK, RST => RST, DOUT => COD_5_OUT);

  cod_4_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_4_IN, CLK => CLK, RST => RST, DOUT => COD_4_OUT);

  cod_3_reg : REG
```

```
    generic map (N => 4)
    port map (
      DIN => COD_3_IN, CLK => CLK, RST => RST, DOUT => COD_3_OUT);

  cod_2_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_2_IN, CLK => CLK, RST => RST, DOUT => COD_2_OUT);

  cod_1_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_1_IN, CLK => CLK, RST => RST, DOUT => COD_1_OUT);

  cod_0_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_0_IN, CLK => CLK, RST => RST, DOUT => COD_0_OUT);

  enable_reg : REG
    generic map (N => 8)
    port map (
      DIN => ENB_IN, CLK => CLK, RST => RST, DOUT => ENB_OUT);

  value_generate : VAL_GEN
    port map (
      MD_IN => MD, SEL_RAD_IN => SEL_OUT(1), SEL_BIT_IN => SEL_OUT(0),
      ENB_N4 => ENB_OUT(7), ENB_4 => ENB_OUT(6), ENB_N3 => ENB_OUT(5),
      ENB_3 => ENB_OUT(4), ENB_N2 => ENB_OUT(3), ENB_2 => ENB_OUT(2),
      ENB_N1 => ENB_OUT(1), ENB_1 => ENB_OUT(0),
      N_MD4 => N_MD4, MD4 => MD4, N_MD3 => N_MD3, MD3 => MD3,
      N_MD2 => N_MD2, MD2 => MD2, N_MD1 => N_MD1, MD1 => MD1);

end STR;
```

Appendix 4: Structure of Partial Product Generate Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Partial Product Generate Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity REC_4 is
  port (
```

```
    CLK, RST                          : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN            : in  std_logic;
    COD_5_IN, COD_4_IN, COD_3_IN      : in  std_logic_vector( 3 downto 0);
    COD_2_IN, COD_1_IN, COD_0_IN      : in  std_logic_vector( 3 downto 0);
    N_MD4_IN,MD4_IN,N_MD3_IN,MD3_IN: in  std_logic_vector(13 downto 0);
    N_MD2_IN,MD2_IN,N_MD1_IN,MD1_IN: in  std_logic_vector(13 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT          : out std_logic;
    MD_COD5_OUT, MD_COD4_OUT          : out std_logic_vector(12 downto 0);
    MD_COD3_OUT, MD_COD2_OUT          : out std_logic_vector(13 downto 0);
    MD_COD1_OUT, MD_COD0_OUT          : out std_logic_vector(13 downto 0);
    HOT_ONE                          : out std_logic_vector( 5 downto 0));
end REC_4;

architecture STR of REC_4 is

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component MD_COD_GEN
    port (
    COD_5_IN, COD_4_IN, COD_3_IN    : in  std_logic_vector( 3 downto 0);
    COD_2_IN, COD_1_IN, COD_0_IN    : in  std_logic_vector( 3 downto 0);
    N_MD4, MD4, N_MD3, MD3          : in  std_logic_vector(13 downto 0);
    N_MD2, MD2, N_MD1, MD1          : in  std_logic_vector(13 downto 0);
    MD_COD5, MD_COD4                : out std_logic_vector(12 downto 0);
    MD_COD3, MD_COD2                : out std_logic_vector(13 downto 0);
    MD_COD1, MD_COD0                : out std_logic_vector(13 downto 0);
    HOT_ONE                        : out std_logic_vector( 5 downto 0));
end component;

  signal SEL_IN, SEL_OUT         : std_logic_vector( 1 downto 0);
  signal COD_5, COD_4, COD_3     : std_logic_vector( 3 downto 0);
  signal COD_2, COD_1, COD_0     : std_logic_vector( 3 downto 0);
  signal N_MD4, MD4, N_MD3, MD3 : std_logic_vector(13 downto 0);
  signal N_MD2, MD2, N_MD1, MD1 : std_logic_vector(13 downto 0);

begin

  SEL_IN      <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  cod_5_reg : REG
    generic map (N => 4)
    port map (
      DIN => COD_5_IN, CLK => CLK, RST => RST, DOUT => COD_5);
```

```
cod_4_reg : REG
  generic map (N => 4)
  port map (
    DIN => COD_4_IN, CLK => CLK, RST => RST, DOUT => COD_4);

cod_3_reg : REG
  generic map (N => 4)
  port map (
    DIN => COD_3_IN, CLK => CLK, RST => RST, DOUT => COD_3);

cod_2_reg : REG
  generic map (N => 4)
  port map (
    DIN => COD_2_IN, CLK => CLK, RST => RST, DOUT => COD_2);

cod_1_reg : REG
  generic map (N => 4)
  port map (
    DIN => COD_1_IN, CLK => CLK, RST => RST, DOUT => COD_1);

cod_0_reg : REG
  generic map (N => 4)
  port map (
    DIN => COD_0_IN, CLK => CLK, RST => RST, DOUT => COD_0);

n_md4_reg : REG
  generic map (N => 14)
  port map (
    DIN => N_MD4_IN, CLK => CLK, RST => RST, DOUT => N_MD4);

md4_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD4_IN, CLK => CLK, RST => RST, DOUT => MD4);

n_md3_reg : REG
  generic map (N => 14)
  port map (
    DIN => N_MD3_IN, CLK => CLK, RST => RST, DOUT => N_MD3);

md3_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD3_IN, CLK => CLK, RST => RST, DOUT => MD3);

n_md2_reg : REG
  generic map (N => 14)
  port map (
    DIN => N_MD2_IN, CLK => CLK, RST => RST, DOUT => N_MD2);

md2_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD2_IN, CLK => CLK, RST => RST, DOUT => MD2);

n_md1_reg : REG
```

```
    generic map (N => 14)
    port map (
      DIN => N_MD1_IN, CLK => CLK, RST => RST, DOUT => N_MD1);

  md1_reg : REG
    generic map (N => 14)
    port map (
      DIN => MD1_IN, CLK => CLK, RST => RST, DOUT => MD1);

  md_coded_generate : MD_COD_GEN
    port map (
      COD_5_IN => COD_5, COD_4_IN => COD_4, COD_3_IN => COD_3,
      COD_2_IN => COD_2, COD_1_IN => COD_1, COD_0_IN => COD_0,
      N_MD4 => N_MD4, MD4 => MD4, N_MD3 => N_MD3, MD3 => MD3,
      N_MD2 => N_MD2, MD2 => MD2, N_MD1 => N_MD1, MD1 => MD1,
      MD_COD5 => MD_COD5_OUT, MD_COD4 => MD_COD4_OUT,
      MD_COD3 => MD_COD3_OUT, MD_COD2 => MD_COD2_OUT,
      MD_COD1 => MD_COD1_OUT, MD_COD0 => MD_COD0_OUT,
      HOT_ONE => HOT_ONE);

end STR;
```

Appendix 5: Structure of First Carry Save Addition Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- First Carry Save Addition Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity ADD1_1 is
  port (
    CLK, RST                : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN  : in  std_logic;
    MD_COD5_IN, MD_COD4_IN  : in  std_logic_vector(12 downto 0);
    MD_COD3_IN, MD_COD2_IN  : in  std_logic_vector(13 downto 0);
    MD_COD1_IN, MD_COD0_IN  : in  std_logic_vector(13 downto 0);
    HOT_ONE_IN              : in  std_logic_vector( 5 downto 0);
    RESULT1_2               : out std_logic_vector(14 downto 0);
    RESULT1_1, RESULT1_0    : out std_logic_vector(16 downto 0);
    COUT1_3                 : out std_logic_vector(15 downto 0);
    COUT1_2, COUT1_1        : out std_logic_vector(17 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT : out std_logic);
end ADD1_1;
```

```
architecture STR of ADD1_1 is

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component CSA_1
    generic (
      N : in integer);
    port (
      A, B, C  : in  std_logic_vector(N-1 downto 0);
      RESULT1  : out std_logic_vector(N-1 downto 0);
      COUT     : out std_logic_vector(  N downto 0));
  end component;

  signal SEL_IN, SEL_OUT                : std_logic_vector( 1 downto 0);
  signal MD_COD5, MD_COD4               : std_logic_vector(12 downto 0);
  signal MD_COD3, MD_COD2               : std_logic_vector(13 downto 0);
  signal MD_COD1, MD_COD0               : std_logic_vector(13 downto 0);
  signal HOT_ONE                        : std_logic_vector( 5 downto 0);
  signal MD_COD5_READY, MD_COD4_READY : std_logic_vector(14 downto 0);
  signal MD_COD3_READY, MD_COD2_READY : std_logic_vector(16 downto 0);
  signal MD_COD1_READY, MD_COD0_READY : std_logic_vector(16 downto 0);
  signal HOT_3_READY                    : std_logic_vector(14 downto 0);
  signal HOT_2_READY, HOT_1_READY       : std_logic_vector(16 downto 0);

begin

  SEL_IN      <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  md_cod5_reg : REG
    generic map (N => 13)
    port map (
      DIN => MD_COD5_IN, CLK => CLK, RST => RST, DOUT => MD_COD5);

  md_cod4_reg : REG
    generic map (N => 13)
    port map (
      DIN => MD_COD4_IN, CLK => CLK, RST => RST, DOUT => MD_COD4);

  md_cod3_reg : REG
    generic map (N => 14)
    port map (
      DIN => MD_COD3_IN, CLK => CLK, RST => RST, DOUT => MD_COD3);
```

```
md_cod2_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD_COD2_IN, CLK => CLK, RST => RST, DOUT => MD_COD2);

md_cod1_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD_COD1_IN, CLK => CLK, RST => RST, DOUT => MD_COD1);

md_cod0_reg : REG
  generic map (N => 14)
  port map (
    DIN => MD_COD0_IN, CLK => CLK, RST => RST, DOUT => MD_COD0);

hot_one_reg : REG
  generic map (N => 6)
  port map (
    DIN => HOT_ONE_IN, CLK => CLK, RST => RST, DOUT => HOT_ONE);

MD_COD5_READY <= MD_COD5 & "00"
                 when SEL_OUT = "01"
                 else (MD_COD5_READY'range => '0');

MD_COD4_READY <= MD_COD4(12) & MD_COD4(12) & MD_COD4
                 when SEL_OUT = "01"
                 else(MD_COD4_READY'range => '0');

MD_COD3_READY <= "00000" & MD_COD3(8)   & MD_COD3( 8 downto 0) & "00"
                 when SEL_OUT = "00"
                 else '0' & MD_COD3(12) & MD_COD3(12 downto 0) & "00"
                 when SEL_OUT = "01"
                 else (MD_COD3_READY'range => '0')
                 when SEL_OUT = "10"
                 else MD_COD3 & "000"
                 when SEL_OUT = "11"
                 else (MD_COD3_READY'range => '0');

MD_COD2_READY <= "00000" & MD_COD2(8) & MD_COD2(8) & MD_COD2(8)
                 & MD_COD2(8 downto 0)
                 when SEL_OUT = "00"
                 else '0' & MD_COD2(12) & MD_COD2(12) & MD_COD2(12)
                 & MD_COD2(12 downto 0)
                 when SEL_OUT = "01"
                 else "000000"& MD_COD2(9) & MD_COD2(9 downto 0)
                 when SEL_OUT = "10"
                 else MD_COD2(13)&MD_COD2(13)&MD_COD2(13)&MD_COD2
                 when SEL_OUT = "11"
                 else (MD_COD2_READY'range => '0');

MD_COD1_READY <= "00000" & MD_COD1(8) & MD_COD1( 8 downto 0) & "00"
                 when SEL_OUT = "00"
                 else '0' & MD_COD1(12) & MD_COD1(12 downto 0) & "00"
                 when SEL_OUT = "01"
                 else "000"& MD_COD1(9) & MD_COD1( 9 downto 0) &"000"
                 when SEL_OUT = "10"
                 else MD_COD1 & "000"
```

```
                when SEL_OUT = "11"
                else (MD_COD1_READY'range => '0');

MD_COD0_READY <= "00000" & MD_COD0(8) & MD_COD0(8) & MD_COD0(8)
                & MD_COD0(8 downto 0)
                when SEL_OUT = "00"
                else '0' & MD_COD0(12) & MD_COD0(12) & MD_COD0(12)
                & MD_COD0(12 downto 0)
                when SEL_OUT = "01"
                else "000" & MD_COD0(9) & MD_COD0(9) & MD_COD0(9)
                & MD_COD0(9) & MD_COD0(9 downto 0)
                when SEL_OUT = "10"
                else MD_COD0(13)& MD_COD0(13)& MD_COD0(13)& MD_COD0
                when SEL_OUT = "11"
                else (MD_COD0_READY'range => '0');

HOT_3_READY <= (2 => HOT_ONE(5), 0 => HOT_ONE(4), others => '0')
               when SEL_OUT = "01"
               else (HOT_3_READY'range => '0');

HOT_2_READY <= (2 => HOT_ONE(3), 0 => HOT_ONE(2), others => '0')
               when SEL_OUT = "00"
               else (2 => HOT_ONE(3), 0 => HOT_ONE(2), others => '0')
               when SEL_OUT = "01"
               else (0 => HOT_ONE(2), others => '0')
               when SEL_OUT = "10"
               else (3 => HOT_ONE(3), 0 => HOT_ONE(2), others => '0')
               when SEL_OUT = "11"
               else (HOT_2_READY'range => '0');

HOT_1_READY <= (2 => HOT_ONE(1), 0 => HOT_ONE(0), others => '0')
               when SEL_OUT = "00"
               else (2 => HOT_ONE(1), 0 => HOT_ONE(0), others => '0')
               when SEL_OUT = "01"
               else (3 => HOT_ONE(1), 0 => HOT_ONE(0), others => '0')
               when SEL_OUT = "10"
               else (3 => HOT_ONE(1), 0 => HOT_ONE(0), others => '0')
               when SEL_OUT = "11"
               else (HOT_2_READY'range => '0');

csa_13 : CSA_1
  generic map (
    N => 15)
  port map (
    A       => MD_COD5_READY,
    B       => MD_COD4_READY,
    C       => HOT_3_READY,
    RESULT1 => RESULT1_2,
    COUT    => COUT1_3);

csa_12 : CSA_1
  generic map (
    N => 17)
  port map (
    A       => MD_COD3_READY,
    B       => MD_COD2_READY,
    C       => HOT_2_READY,
```

```
      RESULT1 => RESULT1_1,
      COUT    => COUT1_2);

  csa_11 : CSA_1
    generic map (
      N => 17)
    port map (
      A       => MD_COD1_READY,
      B       => MD_COD0_READY,
      C       => HOT_1_READY,
      RESULT1 => RESULT1_0,
      COUT    => COUT1_1);

end STR;
```

Appendix 6: Structure of First Carry Propagate Addition Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- First Carry Propagate Addition Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity ADD1_2 is
  port (
    CLK, RST                    : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN      : in  std_logic;
    RESULT1_2_IN                : in  std_logic_vector(14 downto 0);
    RESULT1_1_IN, RESULT1_0_IN : in  std_logic_vector(16 downto 0);
    COUT1_3_IN                  : in  std_logic_vector(15 downto 0);
    COUT1_2_IN, COUT1_1_IN      : in  std_logic_vector(17 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT    : out std_logic;
    RESULT2_2                   : out std_logic_vector(15 downto 0);
    RESULT2_1, RESULT2_0        : out std_logic_vector(17 downto 0));
end ADD1_2;

architecture STR of ADD1_2 is

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
```

```
  end component;

  component CLA_1
    port (
      A, B : in  std_logic_vector(15 downto 0);
      CIN  : in  std_logic;
      SUM  : out std_logic_vector(15 downto 0);
      COUT : out std_logic);
  end component;

  component FANBIT
    generic (
      N : in integer);
    port (
      A, B : in  std_logic_vector(N-1 downto 0);
      CIN  : in  std_logic;
      S    : out std_logic_vector(N-1 downto 0);
      COUT : out std_logic);
  end component;

  signal SEL_IN, SEL_OUT       : std_logic_vector( 1 downto 0);
  signal RESULT_IN2, RESULT1_2 : std_logic_vector(15 downto 0);
  signal RESULT_IN1, RESULT1_1 : std_logic_vector(17 downto 0);
  signal RESULT_IN0, RESULT1_0 : std_logic_vector(17 downto 0);
  signal COUT1_3               : std_logic_vector(15 downto 0);
  signal COUT1_2, COUT1_1      : std_logic_vector(17 downto 0);
  signal C_2, C_1              : std_logic;
  signal RESULT2_1_TEMP2, RESULT2_1_TEMP1:std_logic_vector(1 downto 0);
  signal RESULT2_0_TEMP2, RESULT2_0_TEMP1:std_logic_vector(1 downto 0);
  signal ZERO, ONE                    : std_logic;
  signal RESULT2_OUT2                 : std_logic_vector(15 downto 0);
  signal RESULT2_OUT1, RESULT2_OUT0 : std_logic_vector(17 downto 0);

begin

  SEL_IN        <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT   <= SEL_OUT(1);
  SEL_BIT_OUT   <= SEL_OUT(0);
  RESULT_IN2    <= RESULT1_2_IN(14) & RESULT1_2_IN;
  RESULT_IN1    <= RESULT1_1_IN(16) & RESULT1_1_IN;
  RESULT_IN0    <= RESULT1_0_IN(16) & RESULT1_0_IN;
  ZERO          <= '0';
  ONE           <= '1';

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  result1_2_reg : REG
    generic map (N => 16)
    port map (
      DIN => RESULT_IN2, CLK => CLK, RST => RST, DOUT => RESULT1_2);

  result1_1_reg : REG
    generic map (N => 18)
    port map (
```

```
    DIN => RESULT_IN1, CLK => CLK, RST => RST, DOUT => RESULT1_1);

result1_0_reg : REG
  generic map (N => 18)
  port map (
    DIN => RESULT_IN0, CLK => CLK, RST => RST, DOUT => RESULT1_0);

cout1_3_reg : REG
  generic map (N => 16)
  port map (
    DIN => COUT1_3_IN, CLK => CLK, RST => RST, DOUT => COUT1_3);

cout1_2_reg : REG
  generic map (N => 18)
  port map (
    DIN => COUT1_2_IN, CLK => CLK, RST => RST, DOUT => COUT1_2);

cout1_1_reg : REG
  generic map (N => 18)
  port map (
    DIN => COUT1_1_IN, CLK => CLK, RST => RST, DOUT => COUT1_1);

cla_13 : CLA_1
  port map (
    A    => RESULT1_2,
    B    => COUT1_3,
    CIN  => ZERO,
    SUM  => RESULT2_OUT2,
    COUT => open);

cla_12 : CLA_1
  port map (
    A    => RESULT1_1(15 downto 0),
    B    => COUT1_2(15 downto 0),
    CIN  => ZERO,
    SUM  => RESULT2_OUT1(15 downto 0),
    COUT => C_2);

fa_21 : FANBIT
  generic map (
    N => 2)
  port map (
    A    => RESULT1_1(17 downto 16),
    B    => COUT1_2(17 downto 16),
    CIN  => ONE,
    S    => RESULT2_1_TEMP2,
    COUT => open);

fa_22 : FANBIT
  generic map (
    N => 2)
  port map (
    A    => RESULT1_1(17 downto 16),
    B    => COUT1_2(17 downto 16),
    CIN  => ZERO,
    S    => RESULT2_1_TEMP1,
    COUT => open);
```

```
RESULT2_OUT1(17 downto 16) <= RESULT2_1_TEMP2 when C_2 = '1' else
                              RESULT2_1_TEMP1;

cla_11 : CLA_1
  port map (
    A    => RESULT1_0(15 downto 0),
    B    => COUT1_1(15 downto 0),
    CIN  => ZERO,
    SUM  => RESULT2_OUT0(15 downto 0),
    COUT => C_1);

fa_11 : FANBIT
  generic map (
    N => 2)
  port map (
    A    => RESULT1_0(17 downto 16),
    B    => COUT1_1(17 downto 16),
    CIN  => ONE,
    S    => RESULT2_0_TEMP2,
    COUT => open);

fa_12 : FANBIT
  generic map (
    N => 2)
  port map (
    A    => RESULT1_0(17 downto 16),
    B    => COUT1_1(17 downto 16),
    CIN  => ZERO,
    S    => RESULT2_0_TEMP1,
    COUT => open);

RESULT2_OUT0(17 downto 16) <= RESULT2_0_TEMP2 when C_1 = '1' else
                              RESULT2_0_TEMP1;


RESULT2_2 <= RESULT2_OUT2;

RESULT2_1 <= "000000"  & RESULT2_OUT1(11 downto 0)
             when SEL_OUT = "00" else
             "00"      & RESULT2_OUT1(15 downto 0)
             when SEL_OUT = "01" else
             "00000000"& RESULT2_OUT1( 9 downto 0)
             when SEL_OUT = "10" else
             RESULT2_OUT1
             when SEL_OUT = "11" else
             RESULT2_1'range => '0');

RESULT2_0 <= "000000"  & RESULT2_OUT0(11 downto 0)
             when SEL_OUT = "00" else
             "00"      & RESULT2_OUT0(15 downto 0)
             when SEL_OUT = "01" else
             "0000"    & RESULT2_OUT0(13 downto 0)
             when SEL_OUT = "10" else
             RESULT2_OUT0
             when SEL_OUT = "11" else
             (RESULT2_0'range => '0');
```

```
end STR;
```

Appendix 7: Structure of Second Carry Save Addition Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Second Carry Save Addition Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity ADD2_1 is
  port (
    CLK, RST                    : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN      : in  std_logic;
    RESULT2_2_IN                : in  std_logic_vector(15 downto 0);
    RESULT2_1_IN, RESULT2_0_IN  : in  std_logic_vector(17 downto 0);
    RESULT3                     : out std_logic_vector(22 downto 0);
    COUT                        : out std_logic_vector(22 downto 0);
    SEL_RAD_OUT, SEL_BIT_OUT    : out std_logic);
end ADD2_1;

architecture STR of ADD2_1 is

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;

  component CSA_1
    generic (
      N : in integer);
    port (
      A, B, C  : in  std_logic_vector(N-1 downto 0);
      RESULT1  : out std_logic_vector(N-1 downto 0);
      COUT     : out std_logic_vector(  N downto 0));
  end component;

  signal SEL_IN, SEL_OUT       : std_logic_vector( 1 downto 0);
  signal RESULT2_2             : std_logic_vector(15 downto 0);
  signal RESULT2_1, RESULT2_0  : std_logic_vector(17 downto 0);
```

```
  signal RESULT2_2_READY      : std_logic_vector(22 downto 0);
  signal RESULT2_1_READY      : std_logic_vector(22 downto 0);
  signal RESULT2_0_READY      : std_logic_vector(22 downto 0);
  signal COUT_TEMP            : std_logic_vector(23 downto 0);

begin

  SEL_IN      <= SEL_RAD_IN & SEL_BIT_IN;
  SEL_RAD_OUT <= SEL_OUT(1);
  SEL_BIT_OUT <= SEL_OUT(0);

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  result2_2_reg : REG
    generic map (N => 16)
    port map (
      DIN => RESULT2_2_IN, CLK => CLK, RST => RST, DOUT => RESULT2_2);

  result2_1_reg : REG
    generic map (N => 18)
    port map (
      DIN => RESULT2_1_IN, CLK => CLK, RST => RST, DOUT => RESULT2_1);

  result2_0_reg : REG
    generic map (N => 18)
    port map (
      DIN => RESULT2_0_IN, CLK => CLK, RST => RST, DOUT => RESULT2_0);

  RESULT2_2_READY <= RESULT2_2(14 downto 0) & "00000000"
                     when SEL_OUT = "01" else
                     (RESULT2_2_READY'range => '0');

  RESULT2_1_READY <= "00000000" & RESULT2_1(10 downto 0) & "0000"
                     when SEL_OUT = "00" else
                     RESULT2_1(15) & RESULT2_1(15) & RESULT2_1(15)
                     & RESULT2_1(15 downto 0) & "0000"
                     when SEL_OUT = "01" else
                     "00000000" & RESULT2_1( 8 downto 0) & "000000"
                     when SEL_OUT = "10" else
                     RESULT2_1(16 downto 0) & "000000"
                     when SEL_OUT = "11" else
                     (RESULT2_1_READY'range => '0');

  RESULT2_0_READY <= "00000000" & RESULT2_0(11) & RESULT2_0(11)
                     & RESULT2_0(11) & RESULT2_0(11 downto 0)
                     when SEL_OUT = "00" else
                     RESULT2_0(15) & RESULT2_0(15) & RESULT2_0(15)
                     & RESULT2_0(15) & RESULT2_0(15) & RESULT2_0(15)
                     & RESULT2_0(15) & RESULT2_0(15 downto 0)
                     when SEL_OUT = "01" else
                     "00000000"& RESULT2_0(13) & RESULT2_0(13 downto 0)
                     when SEL_OUT = "10" else
                     RESULT2_0(17) & RESULT2_0(17) & RESULT2_0(17)
                     & RESULT2_0(17) & RESULT2_0(17) & RESULT2_0
```

```
                    when SEL_OUT = "11" else
                    (RESULT2_0_READY'range => '0');

  csa_2 : CSA_1
    generic map (
      N => 23)
    port map (
      A       => RESULT2_2_READY,
      B       => RESULT2_1_READY,
      C       => RESULT2_0_READY,
      RESULT1 => RESULT3,
      COUT    => COUT_TEMP);

  COUT <= COUT_TEMP(22 downto 0);

end STR;
```

Appendix 8: Structure of Second Carry Propagate Addition Stage in VHDL Code

```
--------------------------------------------------------------------------------
-- Second Carry Propagate Addition Stage
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity ADD2_2 is
  port (
    CLK, RST                : in  std_logic;
    SEL_RAD_IN, SEL_BIT_IN : in  std_logic;
    RESULT3_IN              : in  std_logic_vector(22 downto 0);
    COUT_IN                 : in  std_logic_vector(22 downto 0);
    RESULT_FINAL            : out std_logic_vector(22 downto 0));
end ADD2_2;

architecture STR of ADD2_2 is

  component REG
    generic (
      N : in integer);
    port (
      DIN      : in  std_logic_vector(N-1 downto 0);
      CLK, RST : in  std_logic;
      DOUT     : out std_logic_vector(N-1 downto 0));
  end component;
```

```
  component CLA_1
    port (
      A, B : in  std_logic_vector(15 downto 0);
      CIN  : in  std_logic;
      SUM  : out std_logic_vector(15 downto 0);
      COUT : out std_logic);
  end component;

  component FANBIT
    generic (
      N : in integer);
    port (
      A, B : in  std_logic_vector(N-1 downto 0);
      CIN  : in  std_logic;
      S    : out std_logic_vector(N-1 downto 0);
      COUT : out std_logic);
  end component;

  signal SEL_IN, SEL_OUT : std_logic_vector( 1 downto 0);
  signal RESULT3         : std_logic_vector(22 downto 0);
  signal COUT            : std_logic_vector(22 downto 0);
  signal C               : std_logic;
  signal ZERO, ONE       : std_logic;
  signal RESULT_TEMP2, RESULT_TEMP1 : std_logic_vector(6 downto 0);

begin

  SEL_IN <= SEL_RAD_IN & SEL_BIT_IN;
  ZERO   <= '0';
  ONE    <= '1';

  sel_reg : REG
    generic map (N => 2)
    port map (
      DIN => SEL_IN, CLK => CLK, RST => RST, DOUT => SEL_OUT);

  result3_reg : REG
    generic map (N => 23)
    port map (
      DIN => RESULT3_IN, CLK => CLK, RST => RST, DOUT => RESULT3);

  cout_reg : REG
    generic map (N => 23)
    port map (
      DIN => COUT_IN, CLK => CLK, RST => RST, DOUT => COUT);

  cla : CLA_1
    port map (
      A    => RESULT3(15 downto 0),
      B    => COUT(15 downto 0),
      CIN  => ZERO,
      SUM  => RESULT_FINAL(15 downto 0),
      COUT => C);

  fa_2 : FANBIT
    generic map (
      N => 7)
```

```
    port map (
      A    => RESULT3(22 downto 16),
      B    => COUT(22 downto 16),
      CIN  => ONE,
      S    => RESULT_TEMP2,
      COUT => open);

  fa_1 : FANBIT
    generic map (
      N => 7)
    port map (
      A    => RESULT3(22 downto 16),
      B    => COUT(22 downto 16),
      CIN  => ZERO,
      S    => RESULT_TEMP1,
      COUT => open);

  RESULT_FINAL(22 downto 16) <= RESULT_TEMP2 when C = '1' else
                                RESULT_TEMP1;
end STR;
```

Appendix 9: Multiplier Behavior Model in VHDL Code

```
--------------------------------------------------------------------------------
-- Multiplier Behavior Model
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity MULTIPLIER is

  generic (
    N : in integer);
  port (
    MD, MR    : in  std_logic_vector(11 downto 0);
    CLK, RST  : in  std_logic;
    RESULT    : out std_logic_vector(22 downto 0));

end MULTIPLIER;

architecture BEH of MULTIPLIER is
  signal RESULT_LONG1  : std_logic_vector(23 downto 0);
  signal RESULT_LONG2  : std_logic_vector(22 downto 0);
  signal RESULT_SHORT1 : std_logic_vector(15 downto 0);
  signal RESULT_SHORT2 : std_logic_vector(14 downto 0);
```

```
begin

  process(CLK)
    variable MD_SHORT, MR_SHORT : std_logic_vector(7 downto 0);
  begin
    if (CLK'event and CLK = '1') then
      if (RST = '1') then
        RESULT <= (RESULT'range => '0');
      else
        if (N = 8) then
          MD_SHORT      := MD(7 downto 0);
          MR_SHORT      := MR(7 downto 0);
          RESULT_SHORT1 <= signed(MD_SHORT) * signed(MR_SHORT);
          RESULT_SHORT2 <= RESULT_SHORT1(14 downto 0);
          RESULT  <= conv_std_logic_vector(signed(RESULT_SHORT2), 23);
        else
          RESULT_LONG1  <= signed(MD) * signed(MR);
          RESULT_LONG2  <= RESULT_LONG1(22 downto 0);
          RESULT        <= RESULT_LONG2;
        end if;
      end if;
    else
      null;
    end if;
  end process;
end BEH;
```

Appendix 10: Testbench in VHDL Code

```
--------------------------------------------------------------------------------
-- Testbench of the Multiplier
-- Hongfan Wang, 05/25/2000
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity TBMUL is
end TBMUL;

architecture BEH of TBMUL is

  component MUL
    port (
      MD, MR          : in  std_logic_vector(11 downto 0);
      SEL_RAD, SEL_BIT : in  std_logic;
      CLK, RST        : in  std_logic;
      RESULT          : out std_logic_vector(22 downto 0));
```

```
  end component;

  component MULTIPLIER
    generic (
      N : in integer);
    port (
      MD, MR   : in  std_logic_vector(11 downto 0);
      CLK, RST : in  std_logic;
      RESULT   : out std_logic_vector(22 downto 0));
  end component;

  constant N              : integer := 12;
  constant PERIOD         : time    := 20 ns;
  constant STROBE         : time    := PERIOD - 5 ns;
  signal CLK, RST         : std_logic;
  signal SEL_R, SEL_B     : std_logic;
  signal A, B             : std_logic_vector(11 downto 0);
  signal RESULT, RESULT1  : std_logic_vector(22 downto 0);

begin  -- BEH

  SEL_R <= '1';
  SEL_B <= '0' when N = 8 else '1';
  RST   <= '0';

  clk_gen : process
  begin
    CLK <= '0'; wait for PERIOD/2;
    CLK <= '1'; wait for PERIOD/2;
  end process;

  p0 : process
    variable Q : std_logic_vector(N-1 downto 0);
  begin
    A <= (A'range => '1');
    B <= (B'range => '1');
    wait for PERIOD;
    Q := (Q'range => '0');
    loop
      A <= conv_std_logic_vector(signed(Q), 12);
      B <= conv_std_logic_vector(
           signed((Q(N-1) xor (not Q(2))) & Q(N-1 downto 1)), 12);
      wait for PERIOD;
      Q := Q(N-2 downto 0) & (Q(N-1) xor (not Q(2)));
    end loop;
  end process;

  check : process
  begin
    wait for STROBE;
    loop
      case SEL_B is
        when '0' =>
          if (RESULT(14 downto 0) /= RESULT1(14 downto 0)) then
            assert FALSE report "not equal" severity warning;
          end if;
        when '1' =>
```

```
        if (RESULT /= RESULT1) then
          assert FALSE report "not equal" severity warning;
        end if;
      when others => null;
    end case;
    wait for PERIOD;
  end loop;
end process;

mul_ut : MUL
port map (
  MD      => A,
  MR      => B,
  SEL_RAD => SEL_R,
  SEL_BIT => SEL_B,
  CLK     => CLk,
  RST     => RST,
  RESULT  => RESULT);

mul_cal : MULTIPLIER
  generic map (
    N => N)
  port map (
    MD     => A,
    MR     => B,
    CLK    => CLK,
    RST    => RST,
    RESULT => RESULT1);

end BEH;
```

Appendix 11: Synthesized Schematic of Code Generate Stage (REC_1)

CLK: *clock signal*

MD_IN[11:0]: *multiplicand*

RST: *reset signal*

MR_IN[11:0]: *multiplier*

SEL_BIT_IN: *control signal for selecting 8-bit or 12-bit operand wordlength*

SEL_RAD_IN: *control signal for selecting radix-4 or radix-8 recoding scheme*

MD_OUT[11:0]: *multiplicand*

COD_i[3:0]: *code indicating the partial product that needs to be produced, i = 0,1, 2, 3, 4, 5*

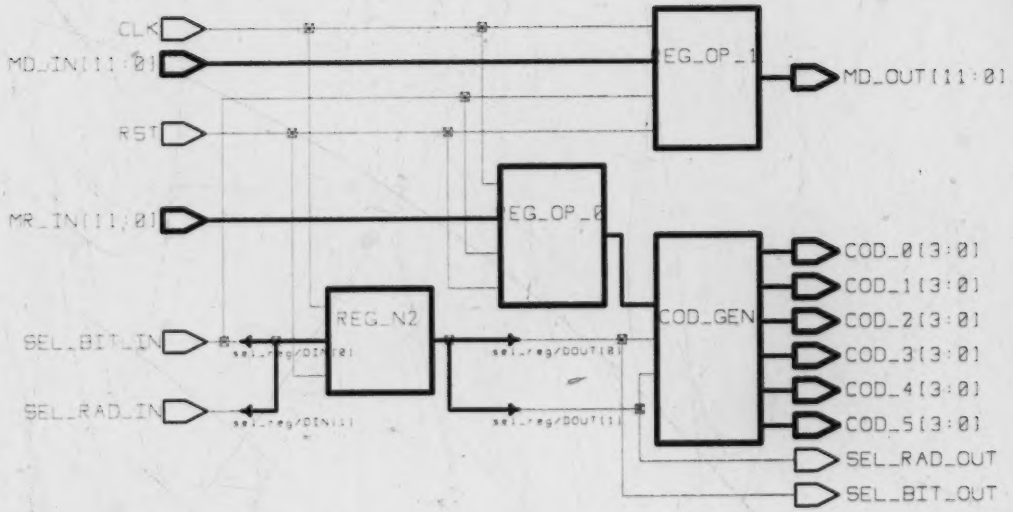
SEL_RAD_OUT: *control signal for selecting radix-4 or radix-8 recoding scheme*

SEL_BIT_OUT: *control signal for selecting 8-bit or 12-bit operand wordlength*

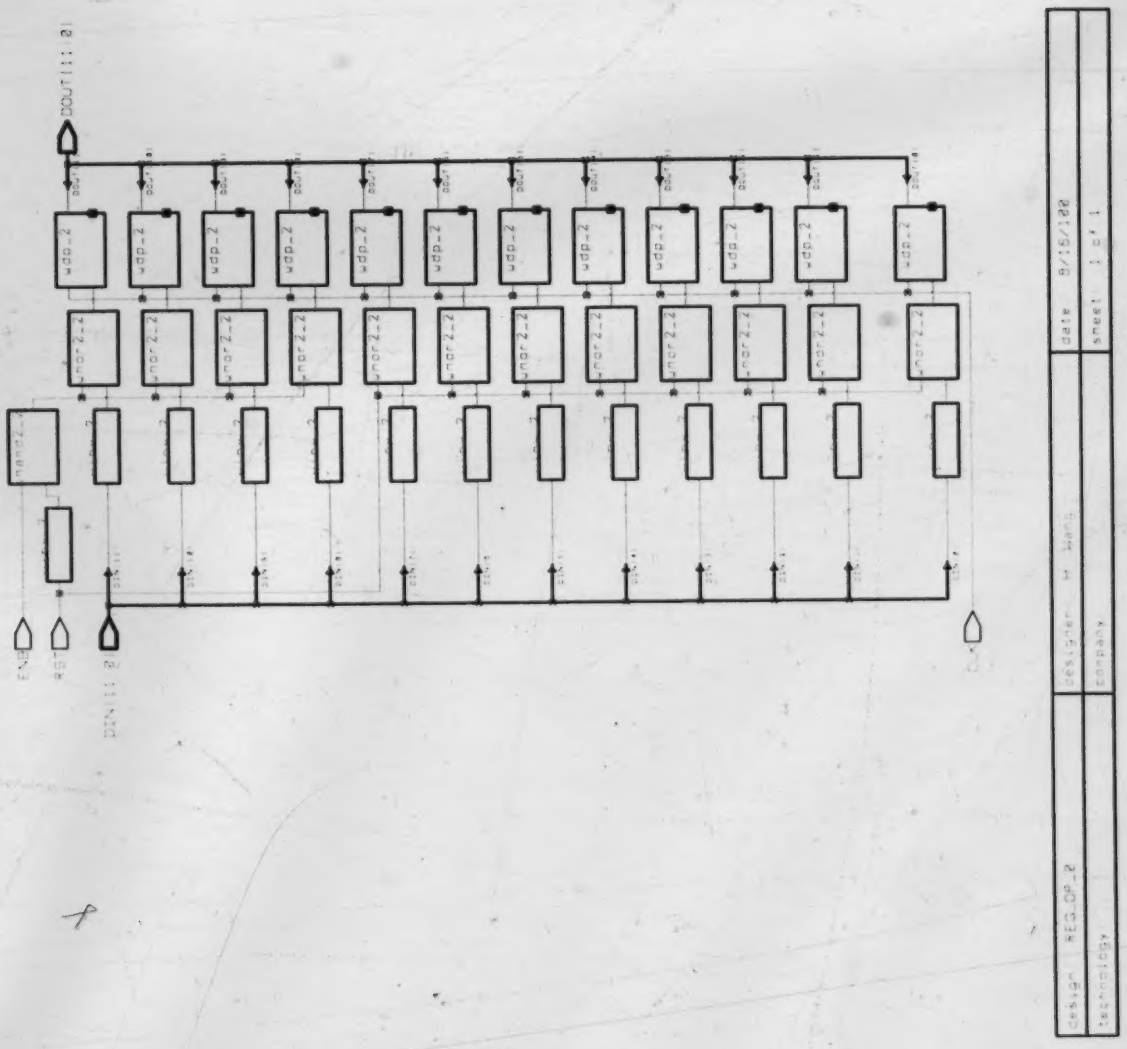
REG_OP_i: *12-bit register, i = 0, 1*

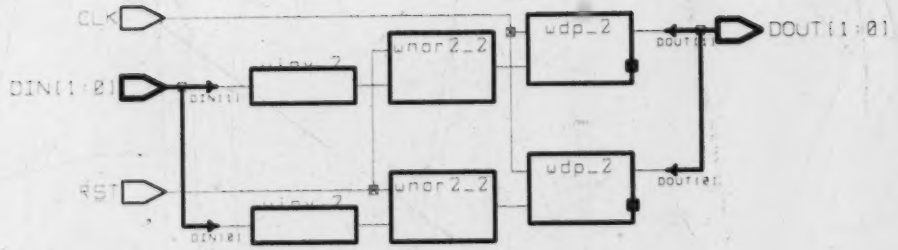
REG_N2: *2-bit register*

COD_GEN: *code generate unit*

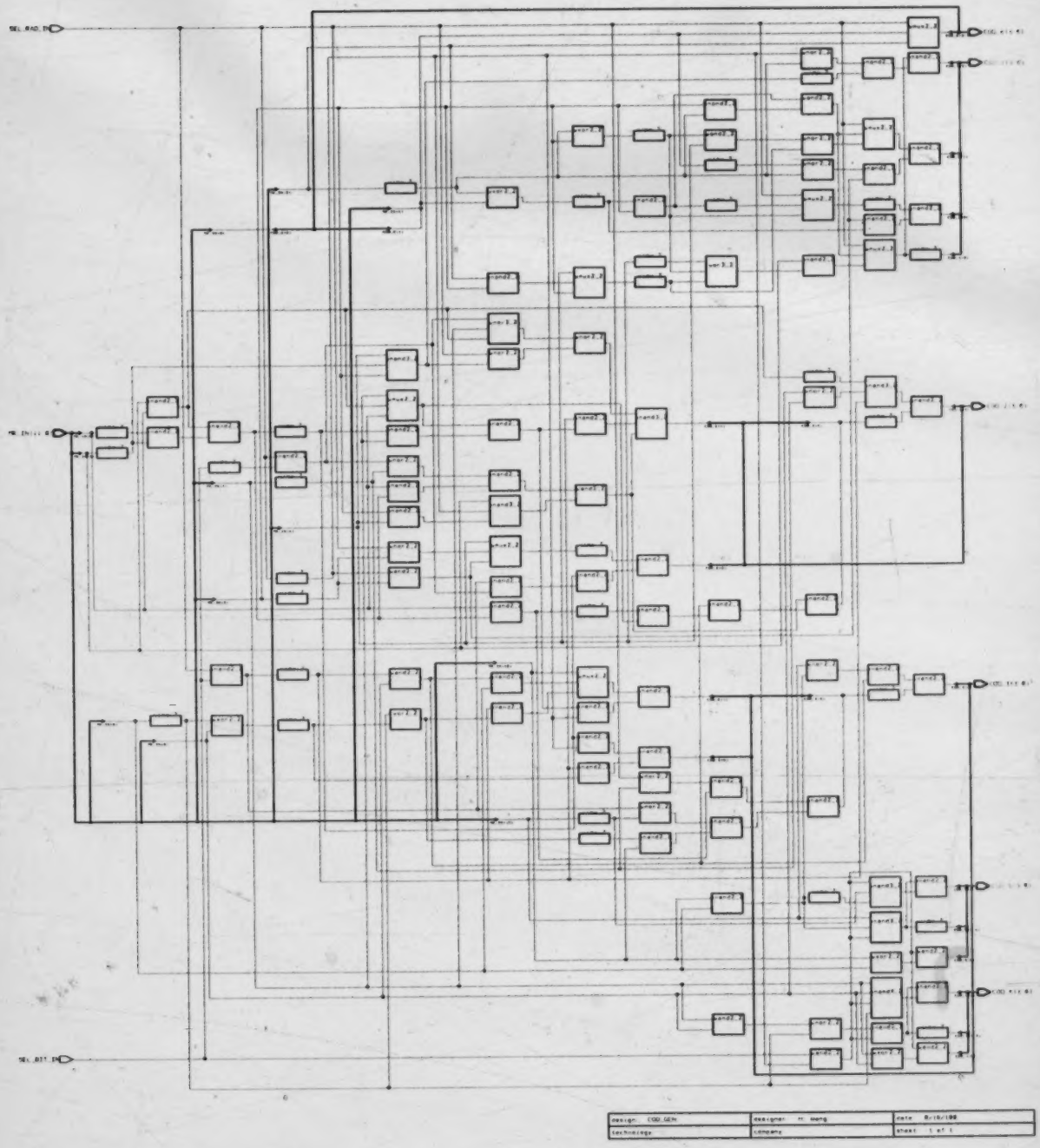


design: REC_1	designer: H. Wang	date: 5/14/100
technology:	company:	sheet: 1 of 1



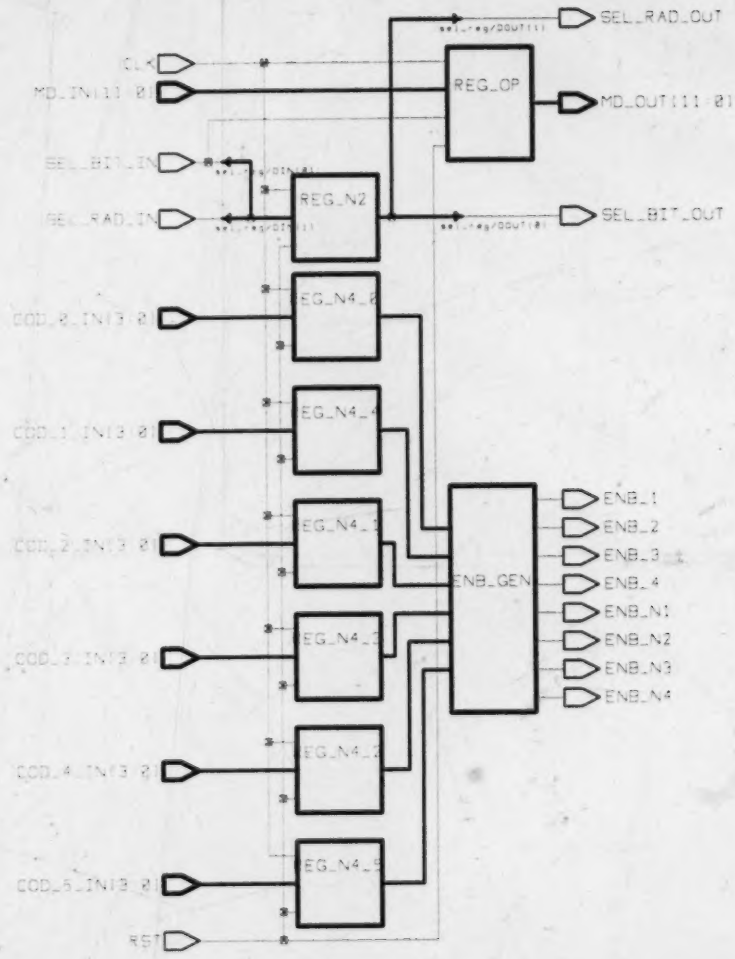


design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1

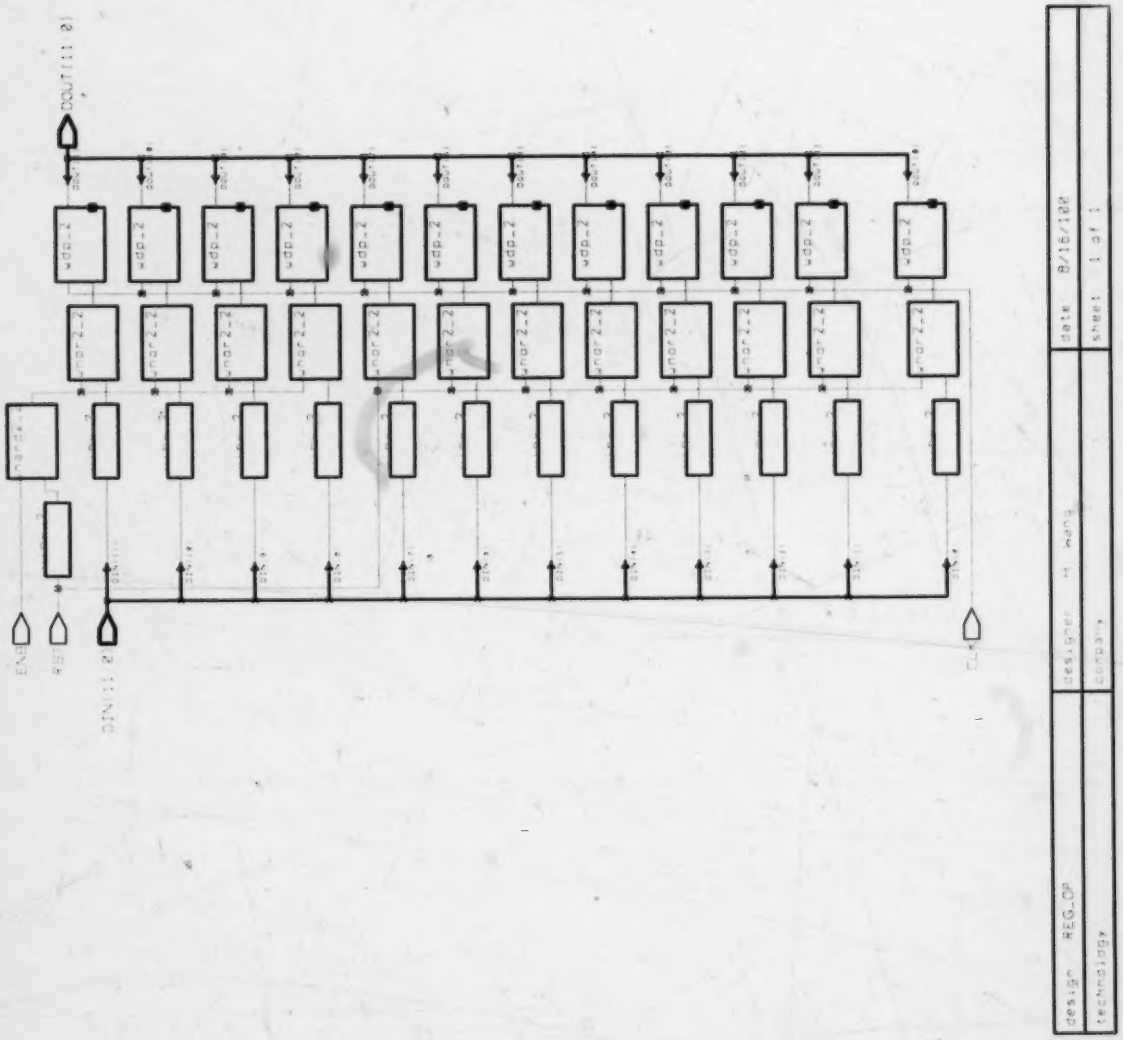


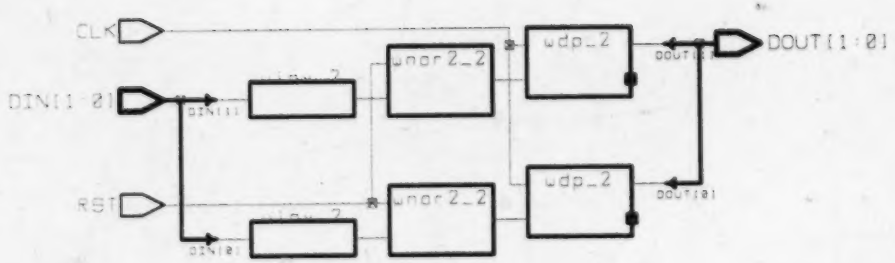
Appendix 12: Synthesized Schematic of Enable Generate Stage (REC_2)

CLK:	*clk signal*
MD_IN[11:0]:	*multiplicand*
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
SEL_RAD_IN:	*control signal for selecting radix-4 or radix-8 recoding scheme*
COD_i_IN[3:0]:	*code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*
RST:	*reset signal*
SEL_RAD_OUT:	*control signal for selecting radix-4 or radix-8 recoding scheme*
MD_OUT[11:0]:	*multiplicand*
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
ENB_i:	*code indicating that i•md is needed later in the calculation, i = 1, n1, 2, n2, 3, n3, 4, n4*
REG_OP:	*12-bit register*
REG_N2:	*2-bit register*
REG_N4_i:	*4-bit register, i = 0, 1, 2, 3, 4, 5*
ENB_GEN:	*enable generate unit*

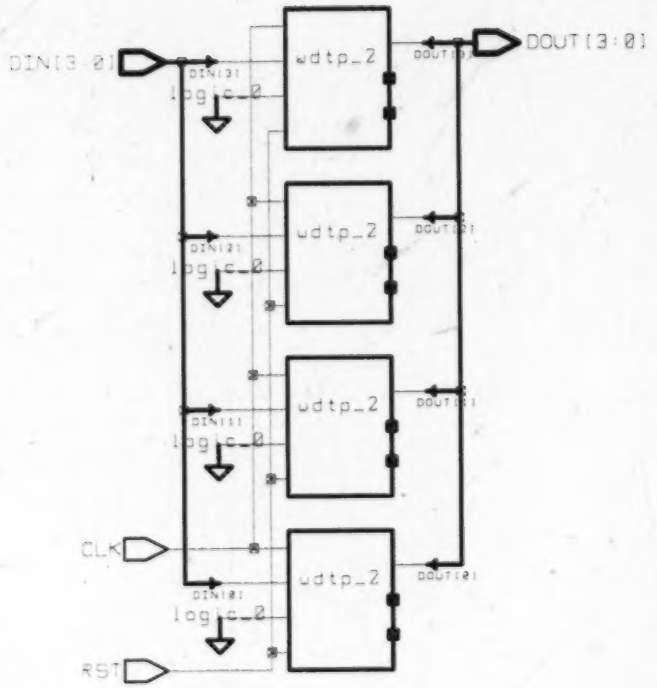


design REC_2	designer H. Wang	date 5/14/100
technology	company	sheet 1 of 1

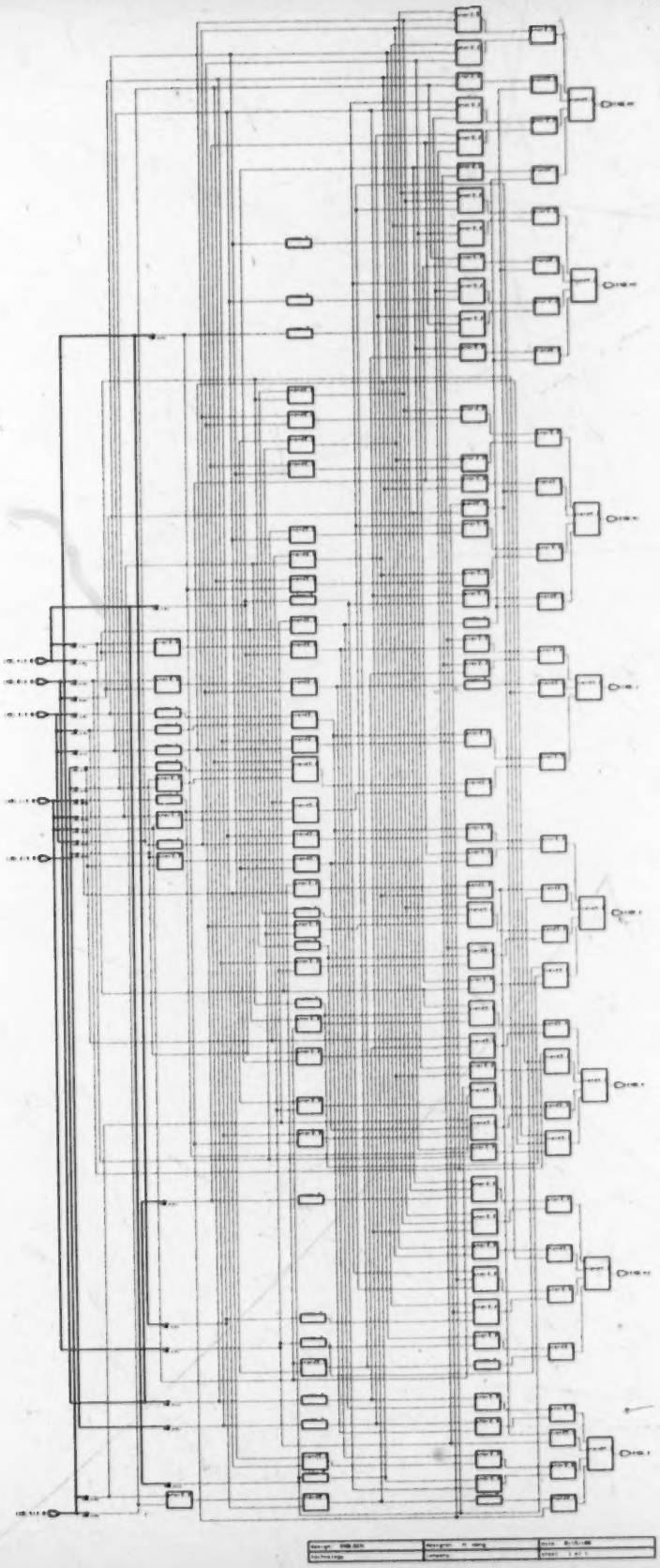

design REG_OP
technology
date 8/16/12M
sheet 1 of 1



design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1

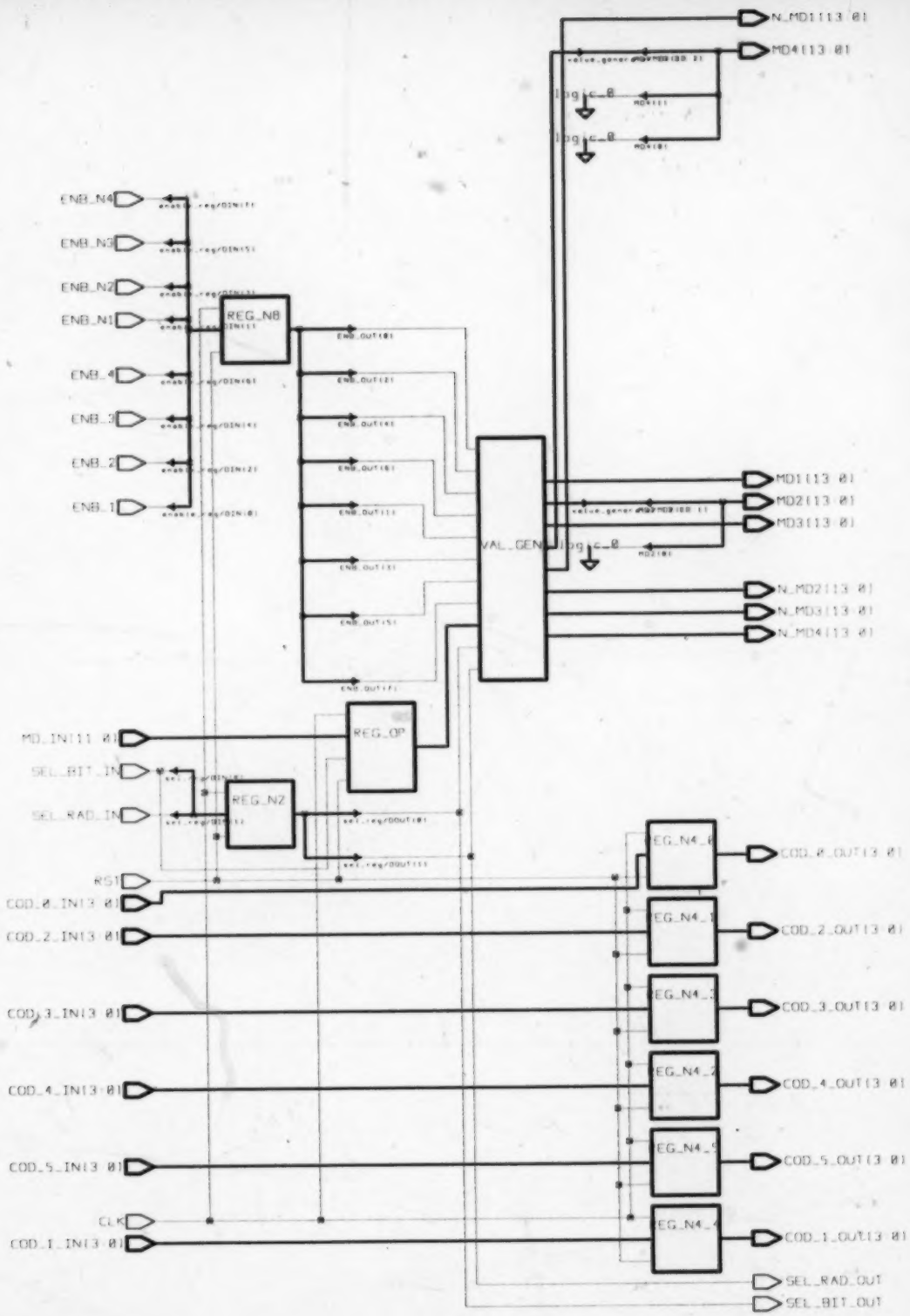


design: REG_N4_0	designer: H. Wang	date: 8/15/100
technology:	company:	sheet: 1 of 1

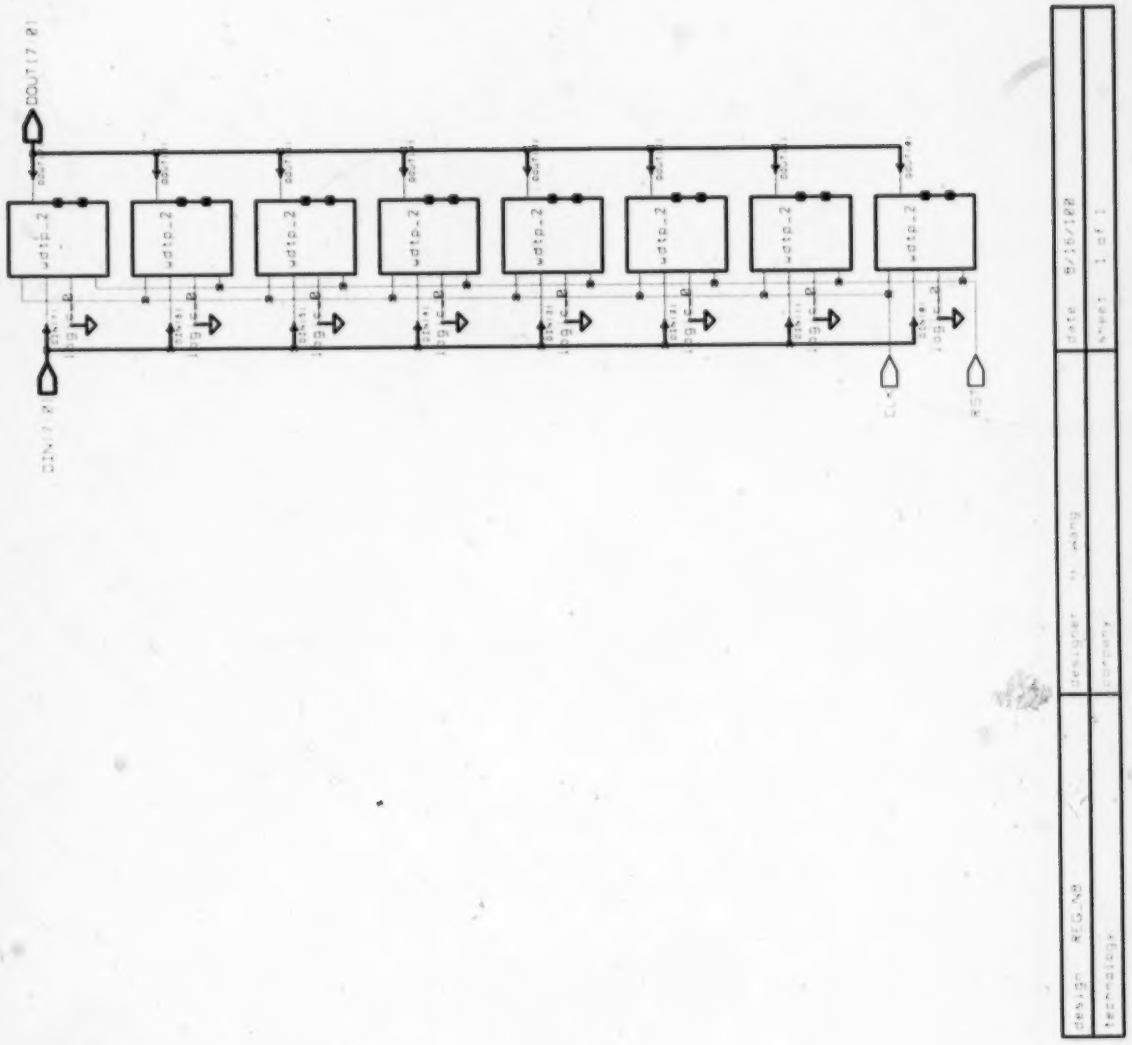


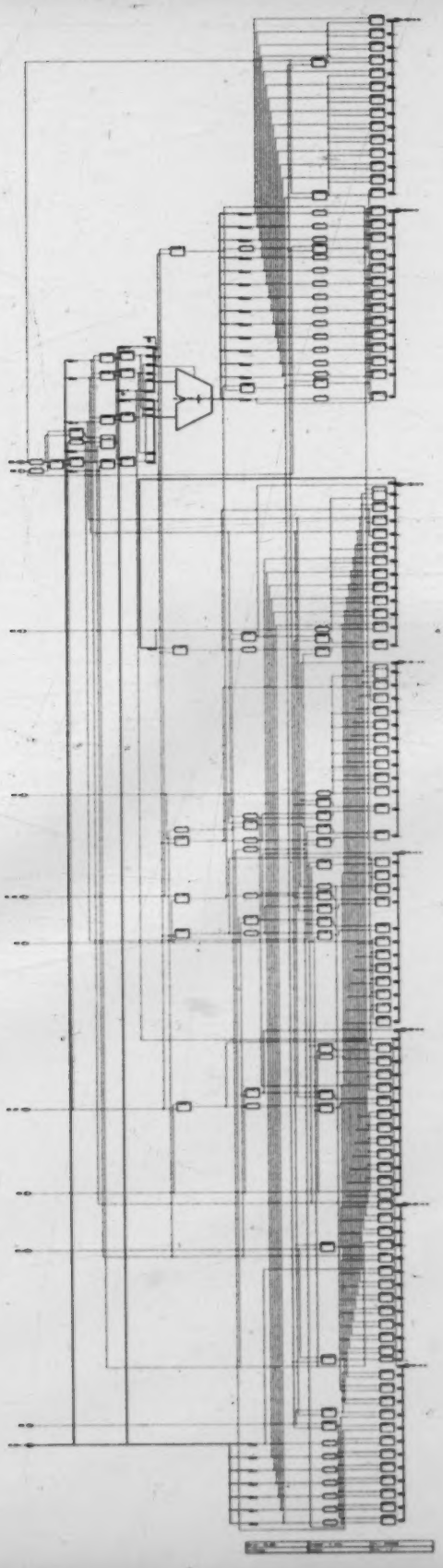
Appendix 13: Synthesized Schematic of Value Generate Stage (REC_3)

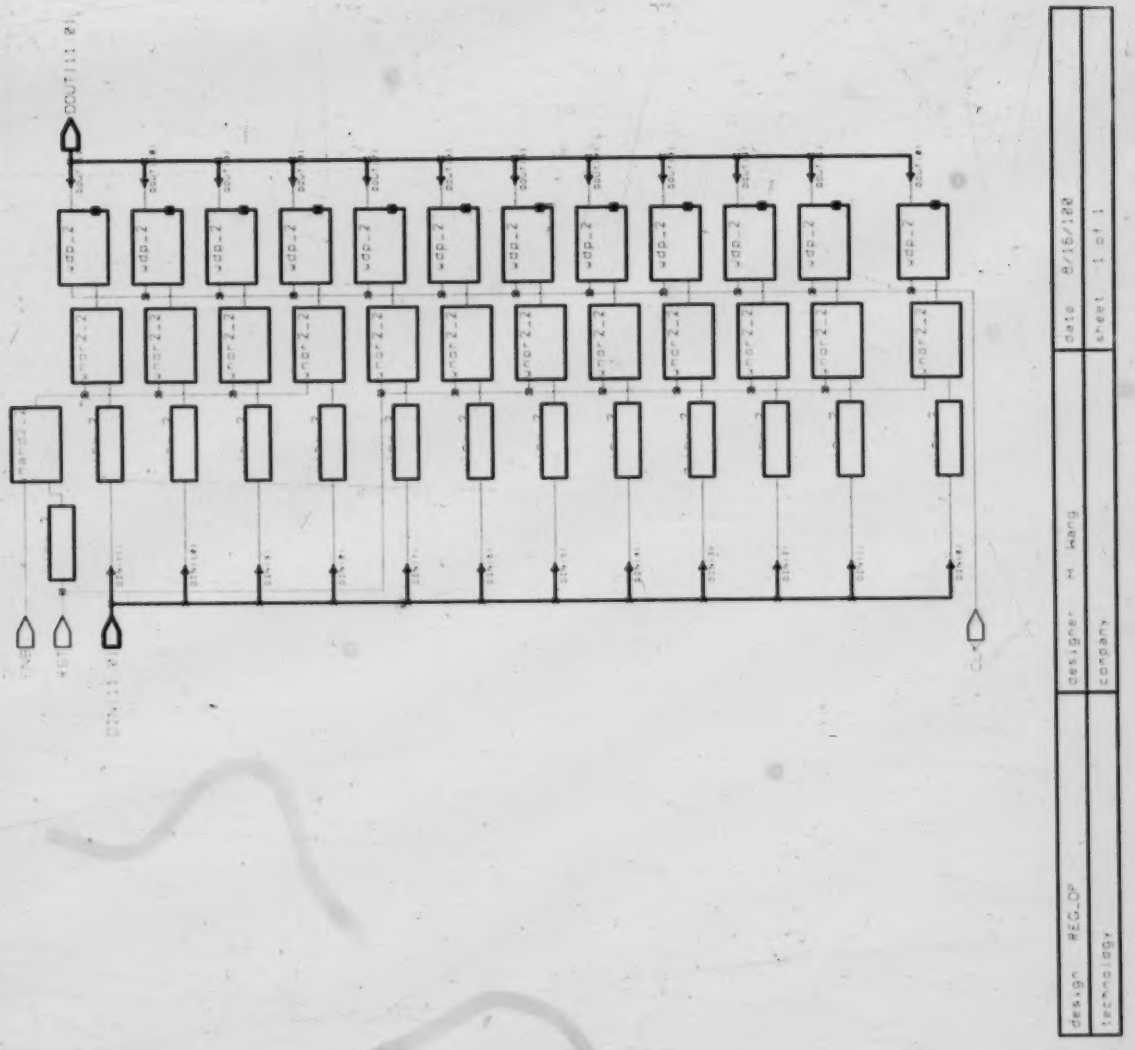
ENB_i:	*code indicating that i•md is needed later in the calculation, i = 1, n1, 2, n2, 3, n3, 4, n4*
MD_IN:	*multiplicand*
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
SEL_RAD_IN:	*control signal for selecting radix-4 or radix-8 recoding scheme*
RST:	*reset signal*
COD_i_IN[3:0]:	*code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*
CLK:	*clock signal*
(N_)Mdi[13:0]:	*value of (-)i•md, i = 1, 2, 3, 4*
COD_i_OUT[3:0]:	*code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*
SEL_RAD_OUT:	*control signal for selecting radix-4 or radix-8 recoding scheme*
SEL_BIT_OUT:	*control signal for selecting 8-bit or 12-bit operand wordlength*
REG_N8:	*8-bit register*
VAL_GEN:	*value generate unit*
REG_OP:	*12-bit register*
REG_N2:	*2-bit register*
REG_N4_i:	*4-bit register, i = 0, 1, 2, 3, 4, 5*

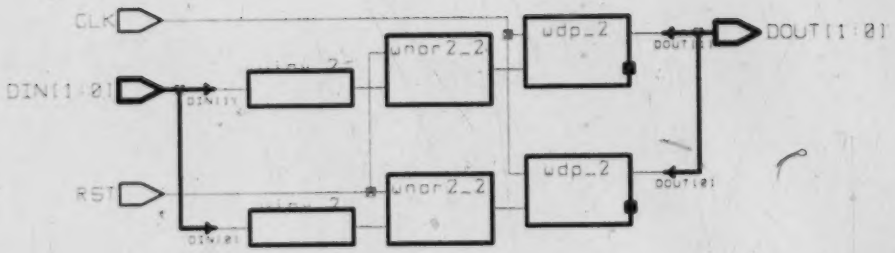


design: REC_3	designer: H. Wang	date: 5/15/100
technology:	company:	sheet: 1 of 1

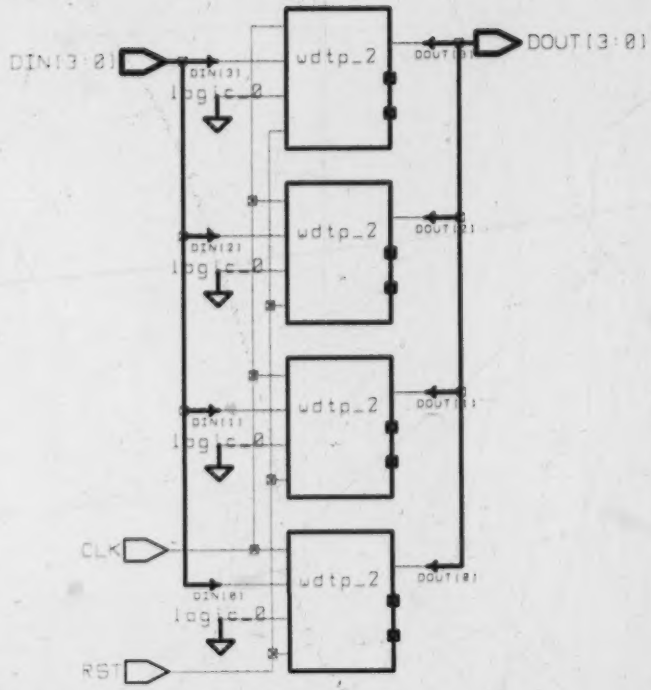





design REG_DP
technology
designer M Wang
company
date 8/15/88
sheet 1 of 1



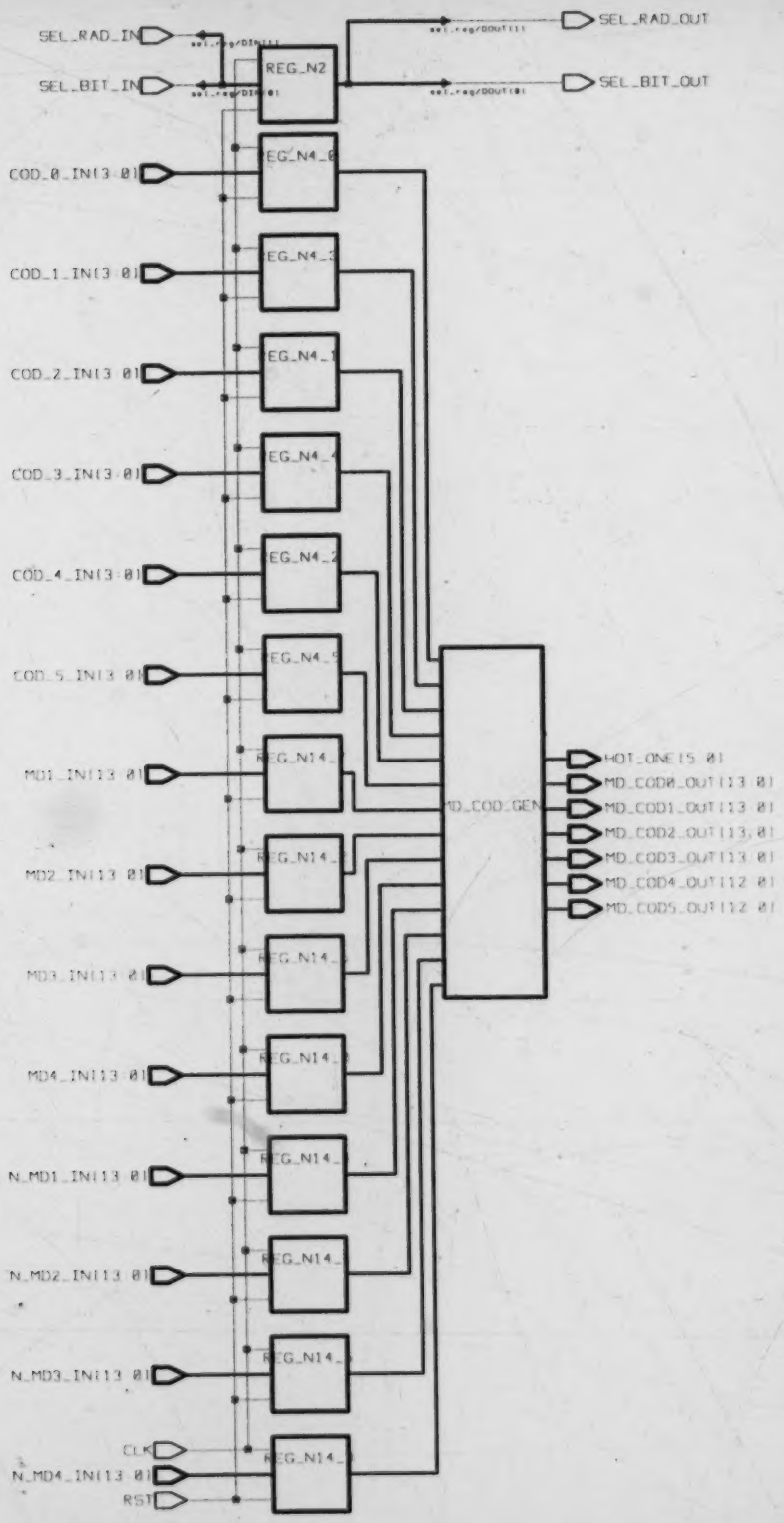
design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



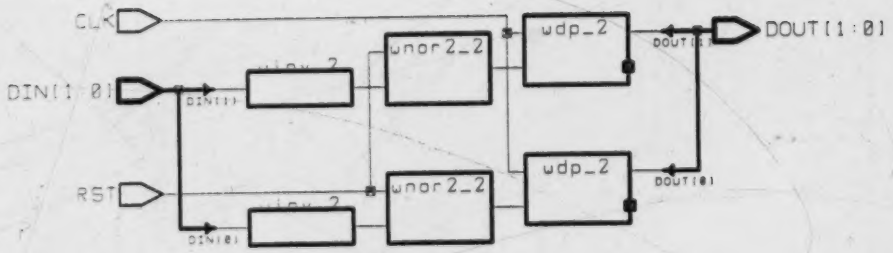
design: REG_N4_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1

Appendix 14: Synthesized Schematic of Partial Product Generate Stage (REC_4)

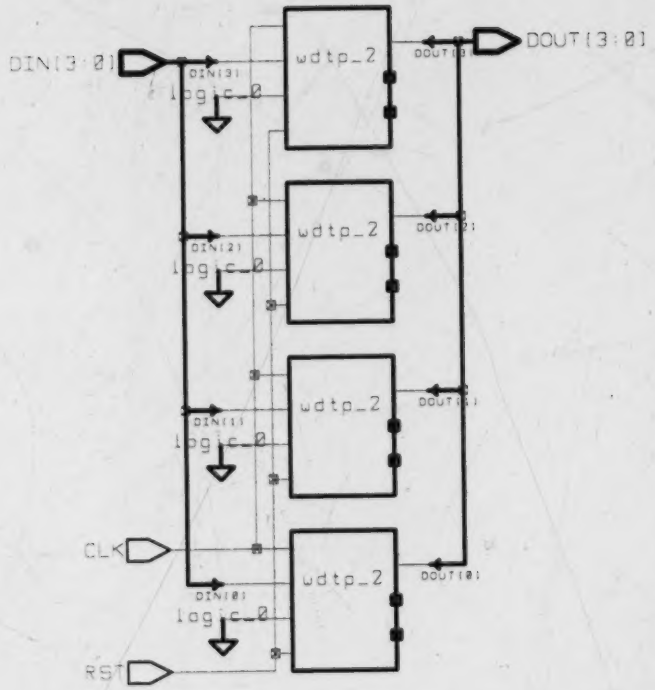
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
SEL_RAD_IN:	*control signal for selecting radix-4 or radix-8 recoding scheme*
COD_i_IN[3:0]:	*code indicating the partial product that needs to be produced, i = 0, 1, 2, 3, 4, 5*
(N_)Mdi[13:0]:	*value of (-)i•md, i = 1, 2, 3, 4*
CLK:	*clock signal*
RST:	*reset signal*
SEL_RAD_OUT:	*control signal for selecting radix-4 or radix-8 recoding scheme*
SEL_BIT_OUT:	*control signal for selecting 8-bit or 12-bit operand wordlength*
HOT_ONE[5:0]:	*code having bitwise correspondence with the sign of each partial product*
MD_CODi_OUT[13:0]:	*ith partial product, i = 0, 1, 2, 3, 4, 5*
MD_CODi_OUT[12:0]:	*ith partial product, i = 4, 5*
REG_N2:	*2-bit register*
REG_N4_i:	*4-bit register, i = 0, 1, 2, 3, 4, 5*
REG_N14_i:	*14-bit register, i = 0, 1, 2, 3, 4, 5*
MD_COD_GEN:	*partial product generate unit, (omit)*



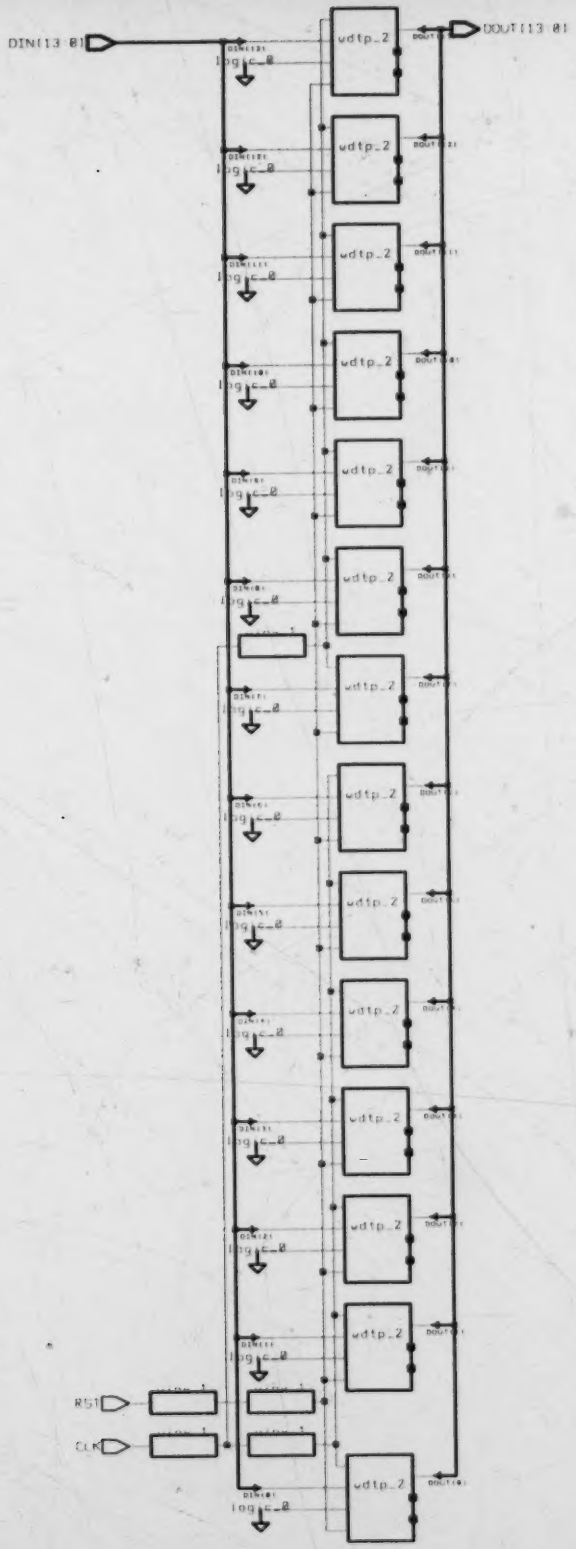
design: REC_4	designer: H. Wang	date: 5/15/100
technology:	company:	sheet: 1 of 1



design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N4_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N14_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1

Appendix 15: Synthesized Schematic of First Carry Save Addition Stage (ADD1_1)

CLK: *clock signal*

RST: *reset signal*

MD_CODi_IN[13:0]: *ith partial product, i = 0, 1, 2, 3*

MD_CODi_IN[12:0]: *ith partial product, i = 4, 5*

HOT_ONE[5:0]: *code having bitwise correspondence with the sign of each partial product*

SEL_BIT_IN: *control signal for selecting 8-bit or 12-bit operand wordlength*

SEL_RAD_IN: *control signal for selecting radix-4 or radix-8 recoding scheme*

RESULT1_2[14:0], RESULT1_1[16:0], RESULT1_0[16:0]: *partial sum*

COUT1_3[15:0], COUT1_2[17:0], COUT1_1[17:0]: *carry-out*

SEL_RAD_OUT: *control signal for selecting radix-4 or radix-8 recoding scheme*

SEL_BIT_OUT: *control signal for selecting 8-bit or 12-bit operand wordlength*

CSA_1_N15: *CSA unit #3*

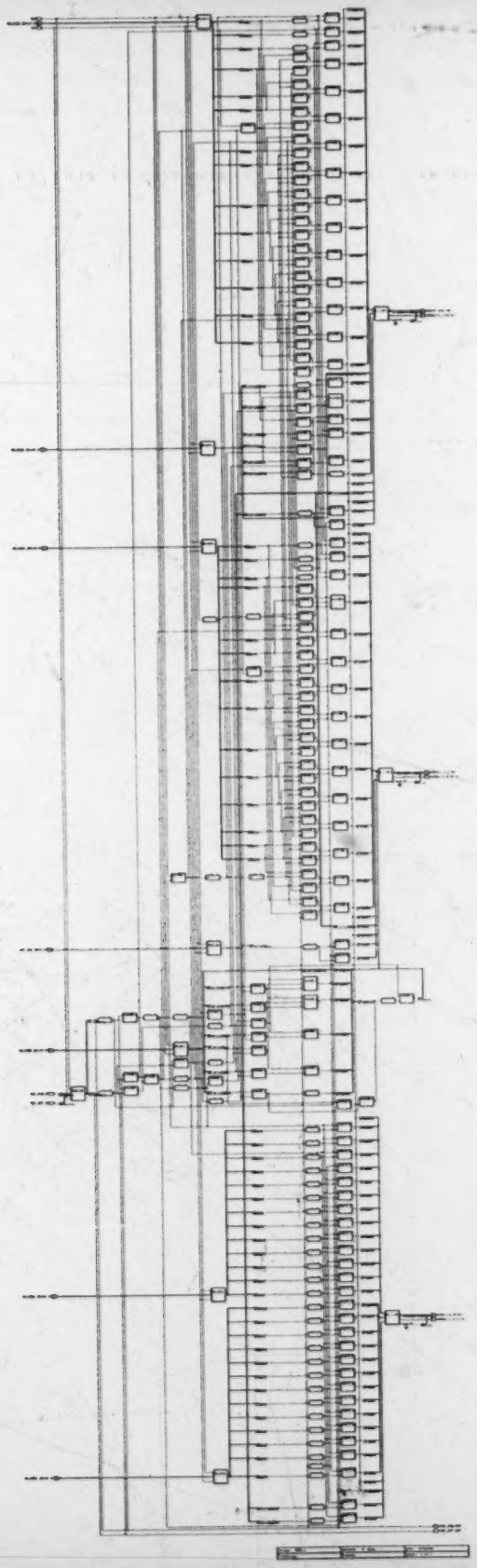
CSA_1_N17_i: *CSA unit #i, i = 1, 2*

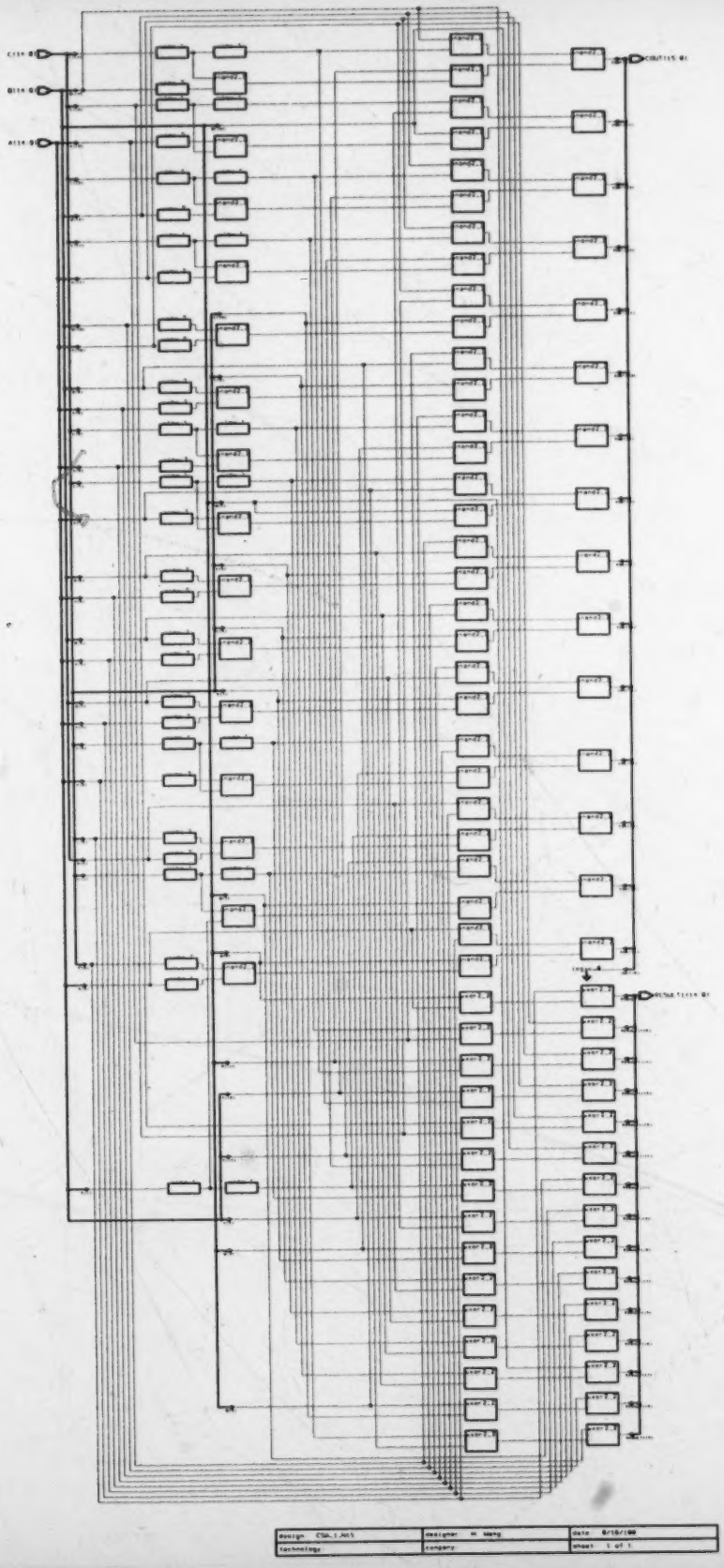
REG_N2: *2-bit register*

REG_N6: *6-bit register*

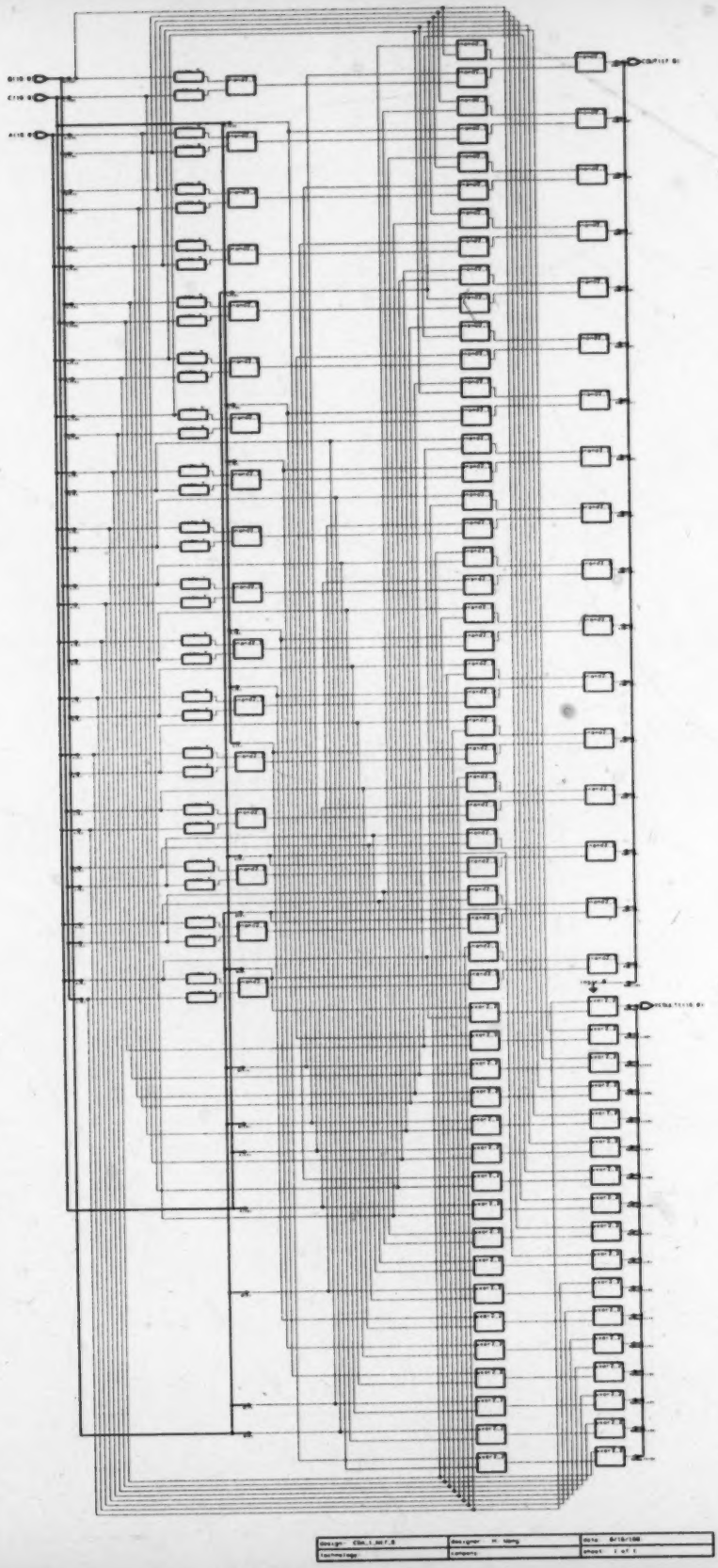
REG_N13_i: *13-bit register, i = 0, 1*

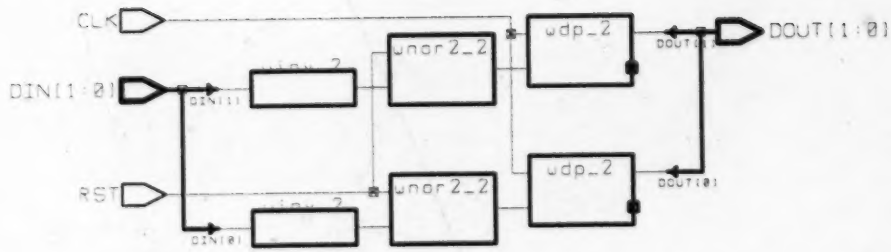
REG_N14_i: *14-bit register, i = 0, 1, 2, 3*



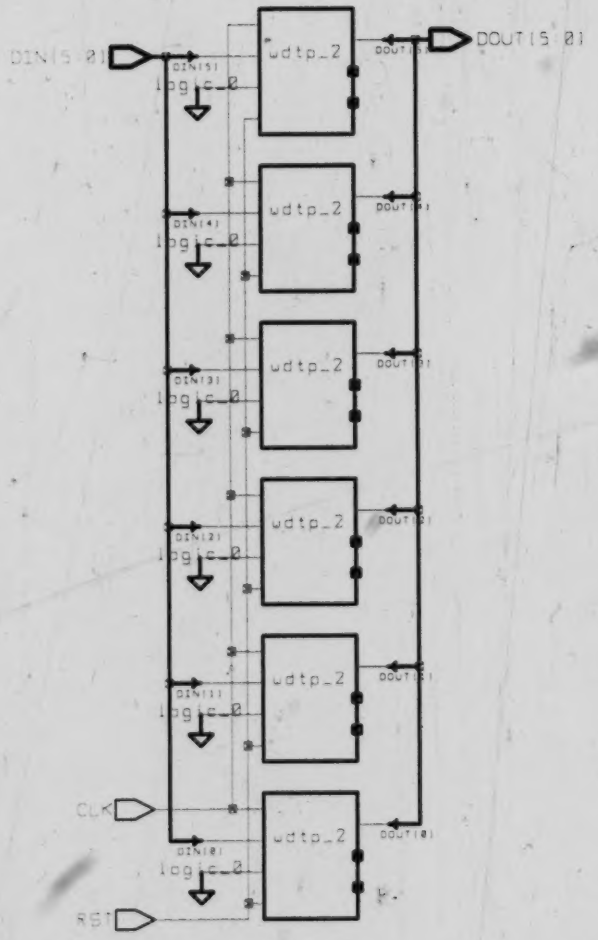


design CSA_1.MS	designer H. Wang	date 6/15/88
technology	company	sheet 1 of 1

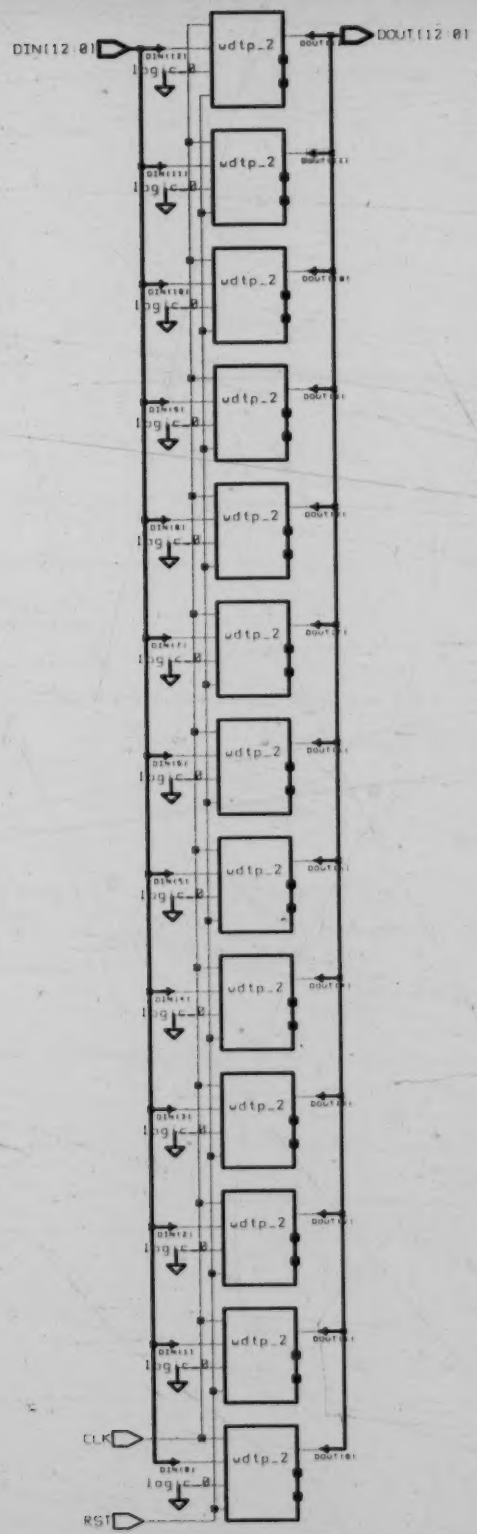




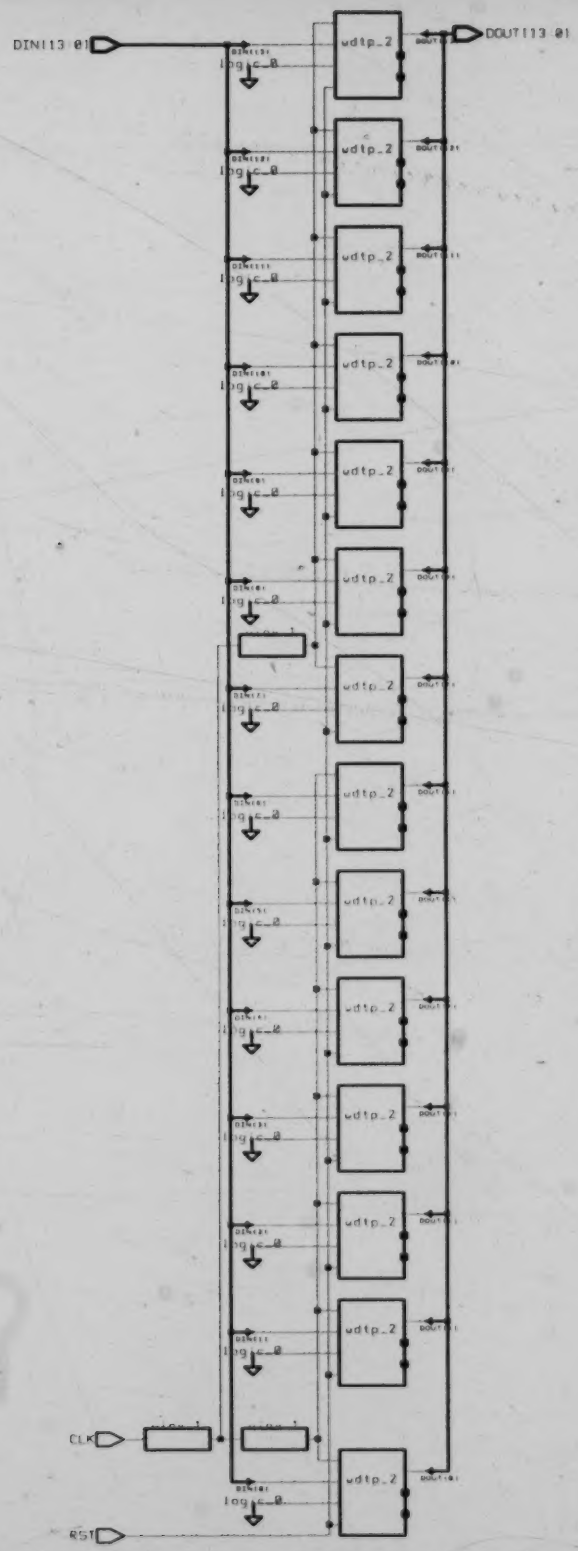
design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N6	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N13_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N14_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet 1 of 1

Appendix 16: Synthesized Schematic of First Carry Propagate Addition Stage (ADD1_2)

RESULT1_2[14:0], RESULT1_1[16:0], RESULT1_0[16:0]: *partial sum*

COUT1_3[15:0], COUT1_2[17:0], COUT1_1[17:0]: *carry-out*

CLK: *clock signal*

RST: *reset signal*

SEL_RAD_IN: *control signal for selecting radix-4 or radix-8 recoding scheme*

SEL_BIT_IN: *control signal for selecting 8-bit or 12-bit operand wordlength*

SEL_RAD_OUT: *control signal for selecting radix-4 or radix-8 recoding scheme*

SEL_BIT_OUT: *control signal for selecting 8-bit or 12-bit operand wordlength*

RESULT2_2[15:0], RESULT2_1[17:0], RESULT2_0[17:0]: *intermediate result*

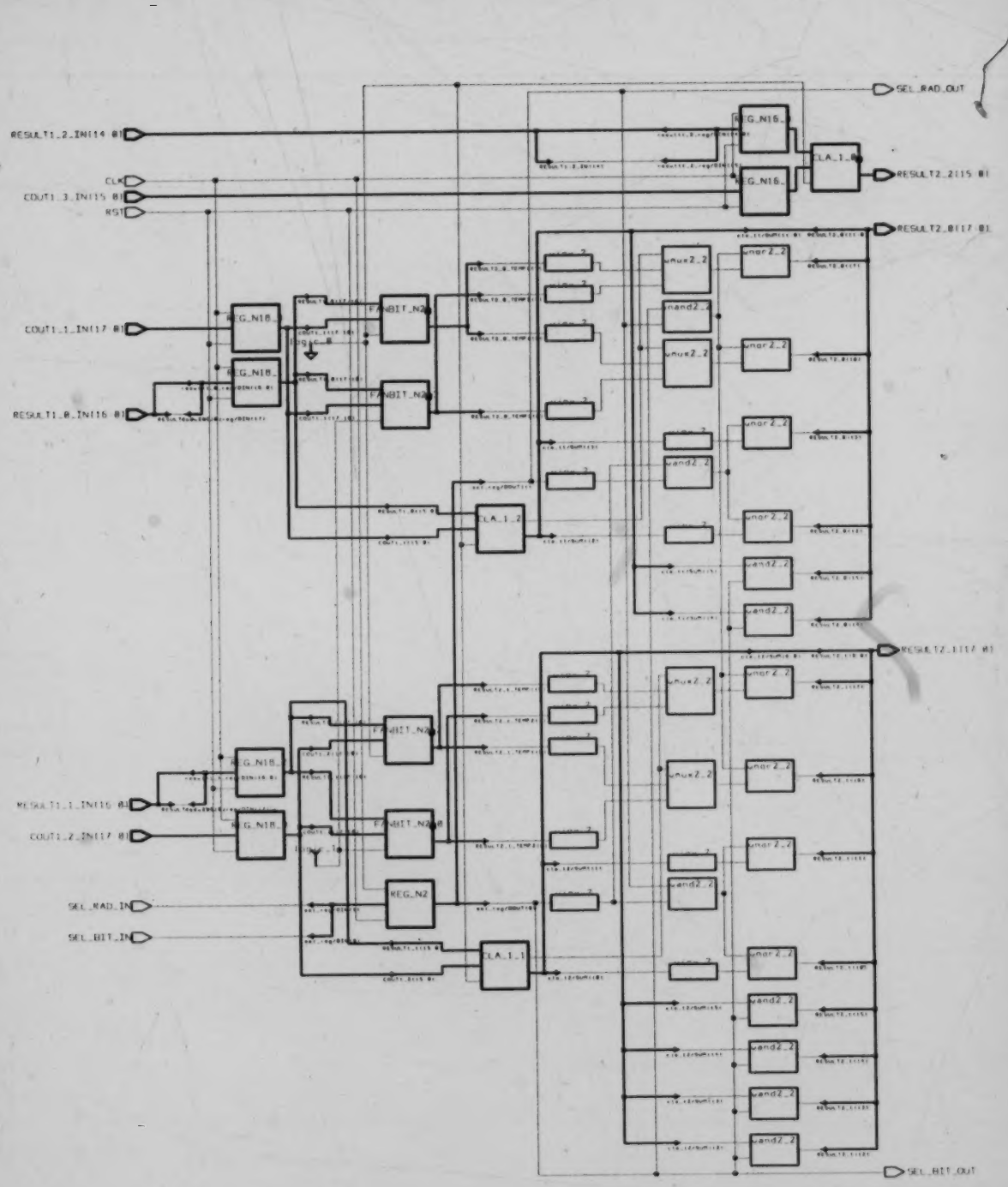
CLA_1_i: *carry lookahead adder, i = 0, 1, 2*

FANBIT_N2_i: *2-bit full adder, i = 0, 1, 2, 3*

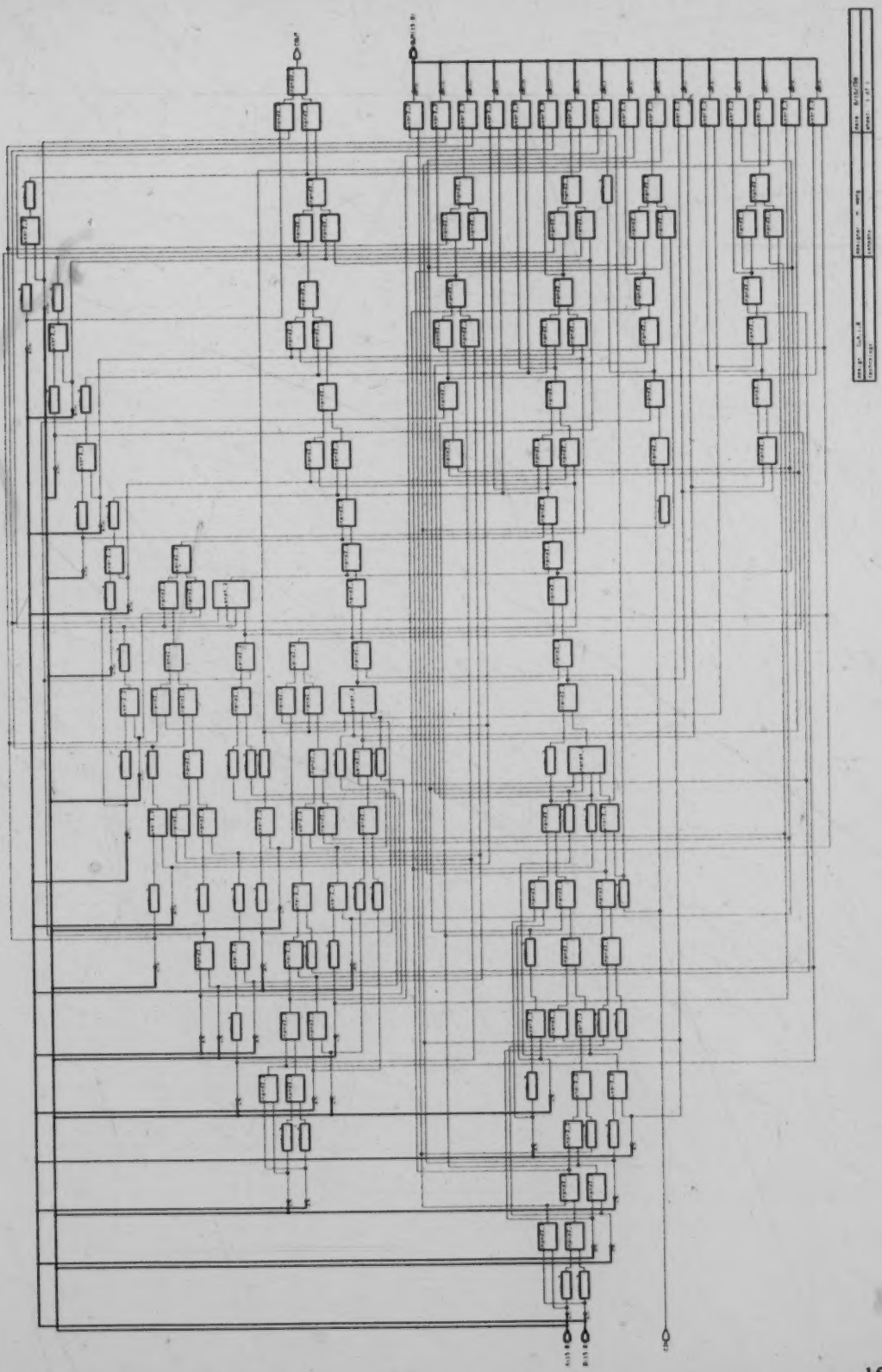
REG_N2: *2-bit register*

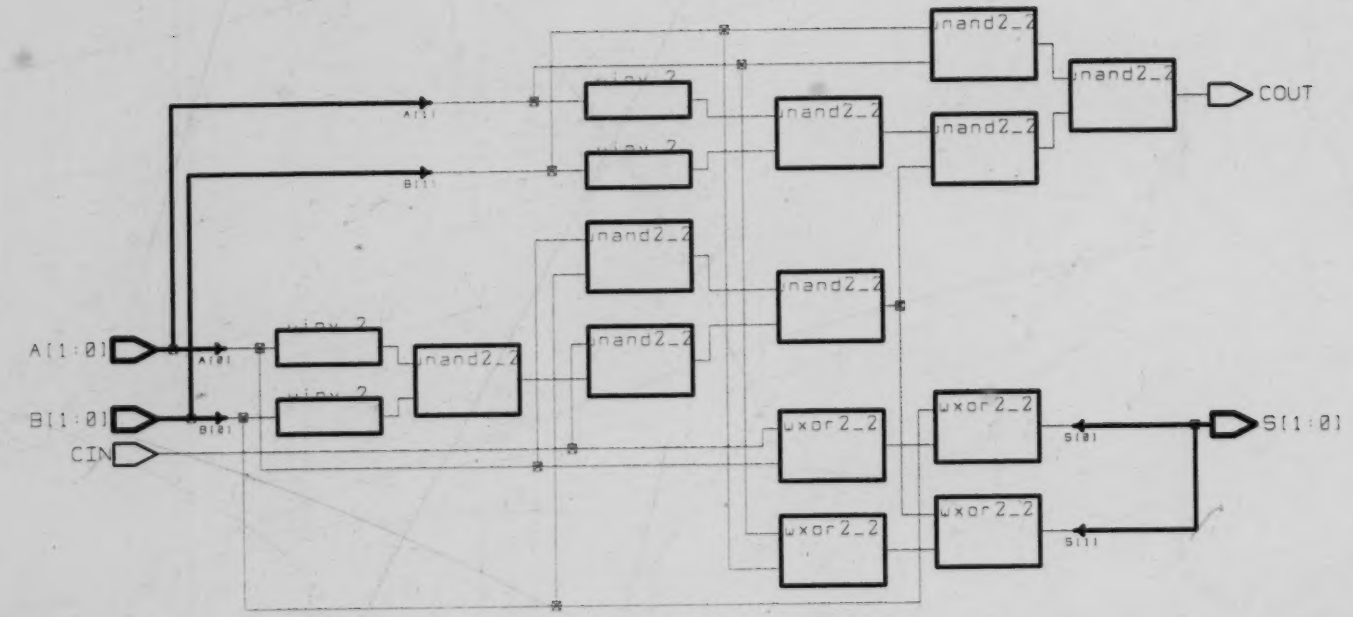
REG_N16_i: *16-bit register, i = 0, 1*

REG_N18_i: *18-bit register, i = 0, 1, 2, 3*

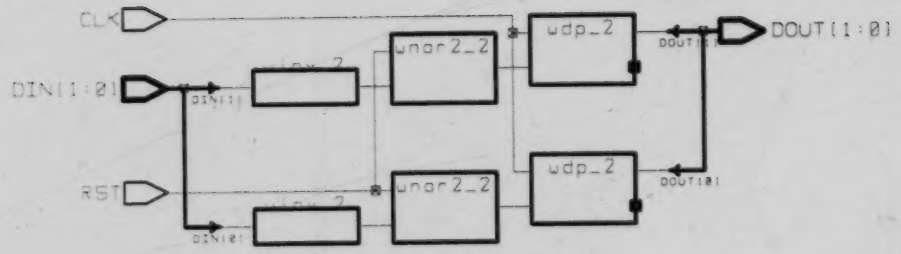


design ADD1_2	designer H Wang	date 8/15/100
technology	company	sheet 1 of 1

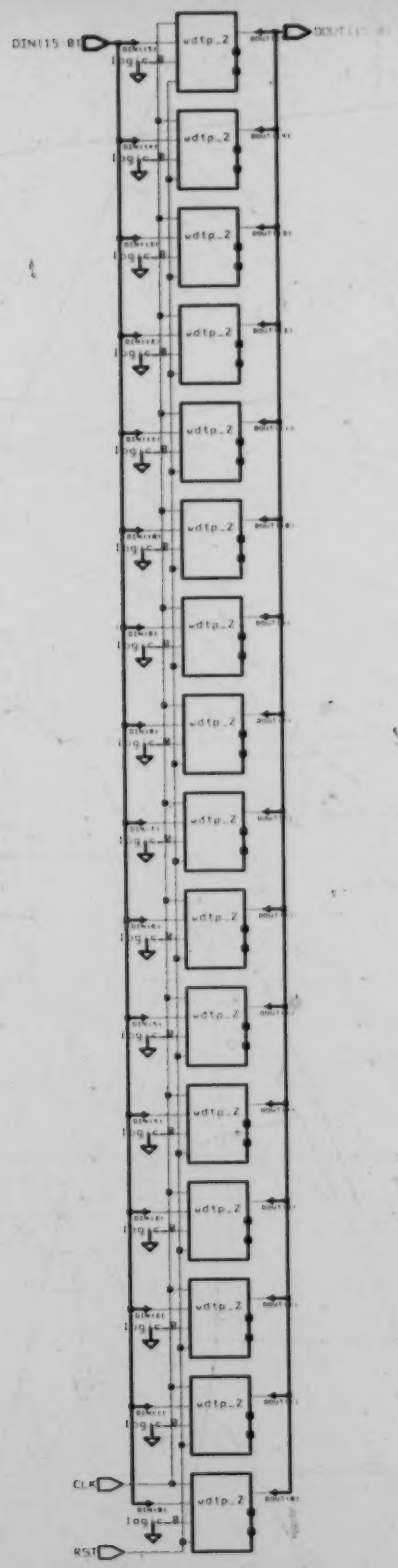




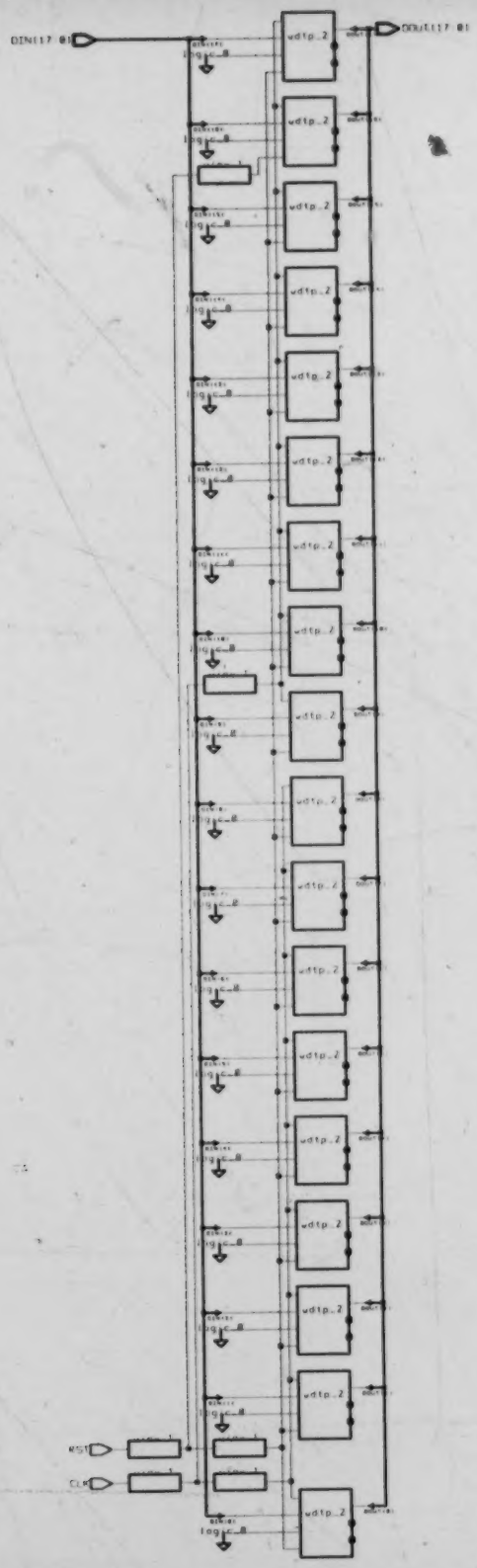
design: FANBIT_N2_0	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design REG_N16_8	designer H. Wang	date 8/16/100
technology	company	sheet 1 of 1

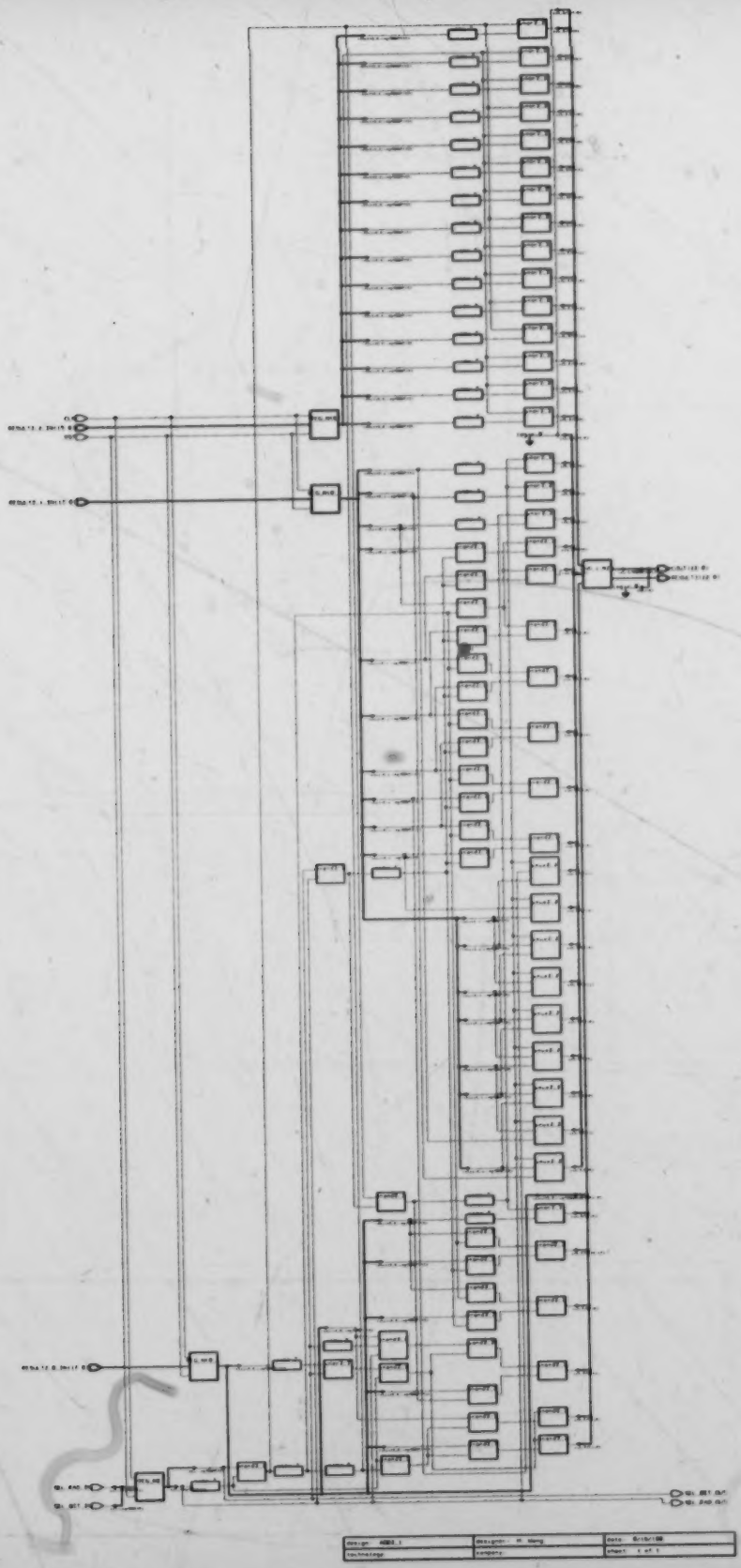


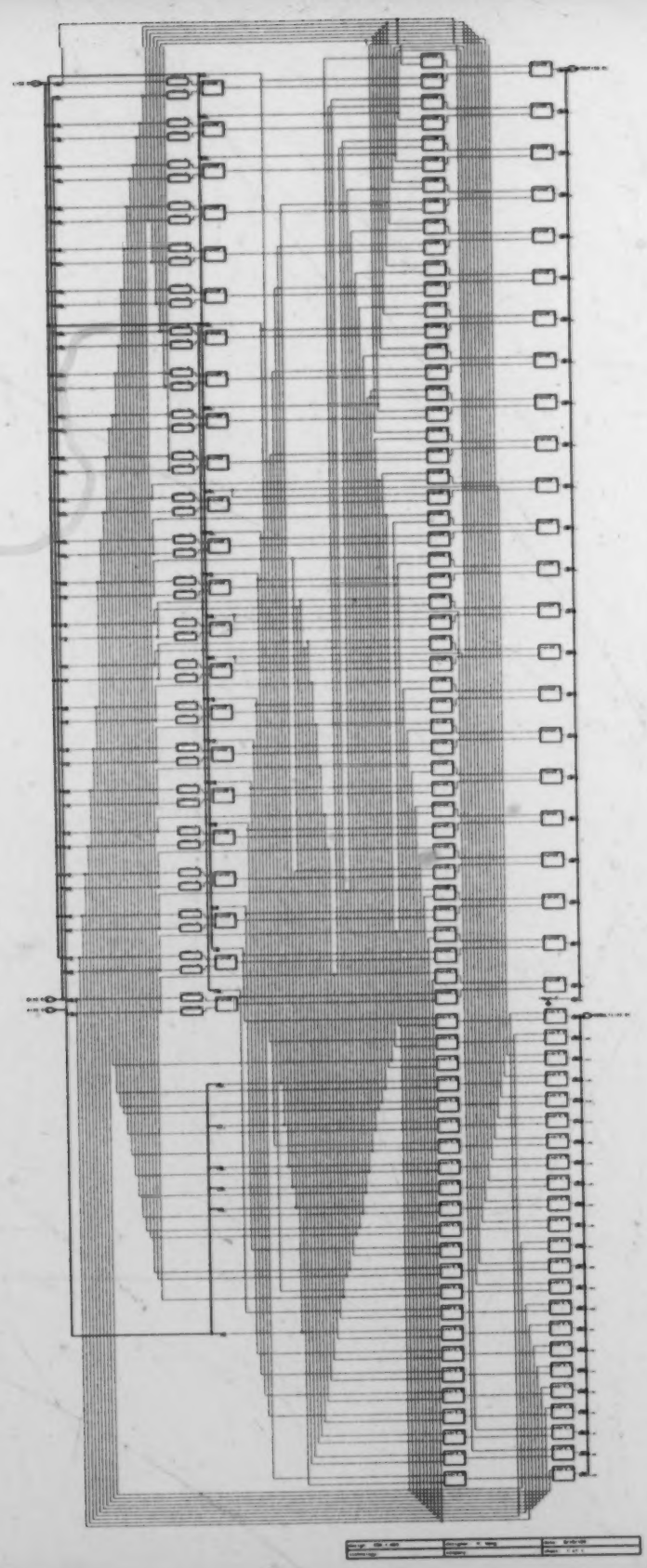
design REG_N18_0	designer H. Wang	date 8/16/100
technology	company -	sheet 1 of 1

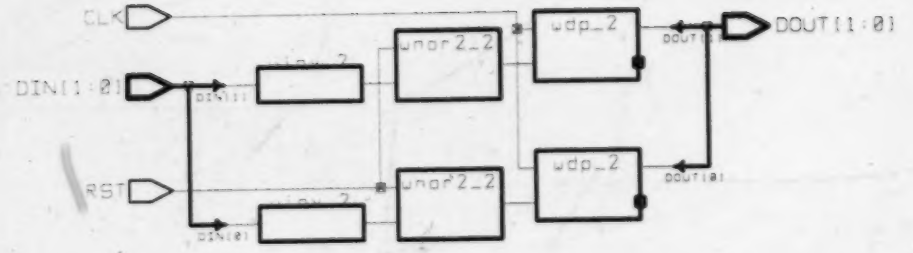
Appendix 17: Synthesized Schematic of Second Carry Save Addition Stage (ADD2_1)

RESULT2_2_IN[15:0], RESULT2_1_IN[17:0], RESULT2_0_IN[16:0]: *intermediate result*

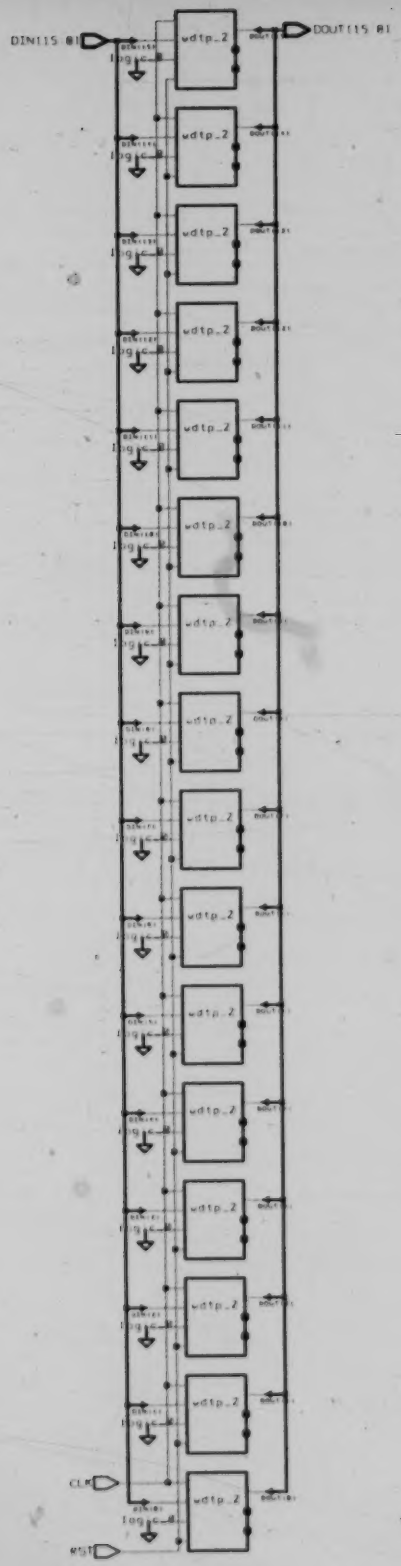
CLK:	*clock signal*
RST:	*reset signal*
SEL_RAD_IN:	*control signal for selecting radix-4 or radix-8 recoding scheme*
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
COUT[22:0]:	*carry-out*
RESULT3[22:0]:	*partial sum*
SEL_BIT_OUT:	*control signal for selecting 8-bit or 12-bit operand wordlength*
SEL_RAD_OUT:	*control signal for selecting radix-4 or radix-8 recoding scheme*
CSA_1_N23:	*CSA unit*
REG_N2:	*2-bit register*
REG_N16:	*16-bit register*
REG_N18_i:	*18-bit register, i = 0, 1*



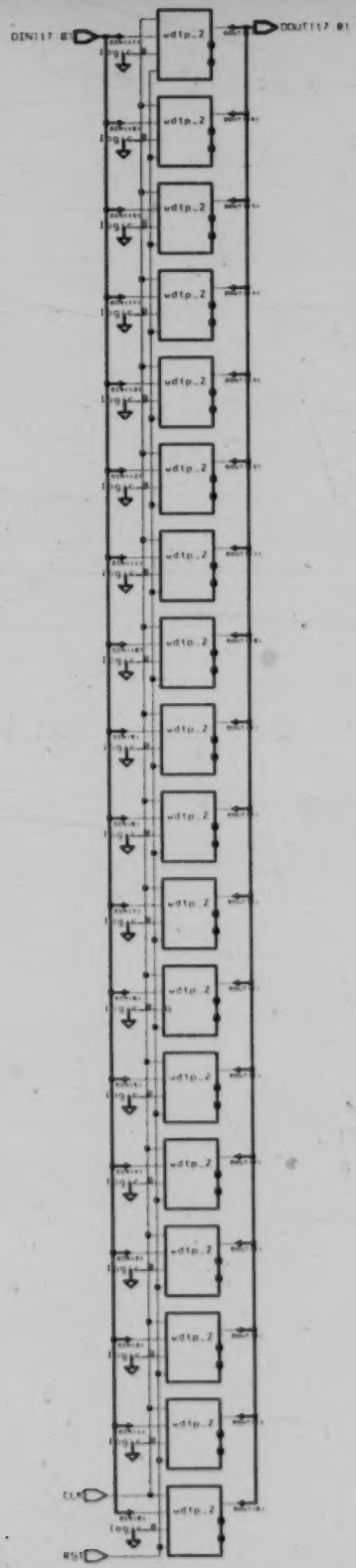




design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



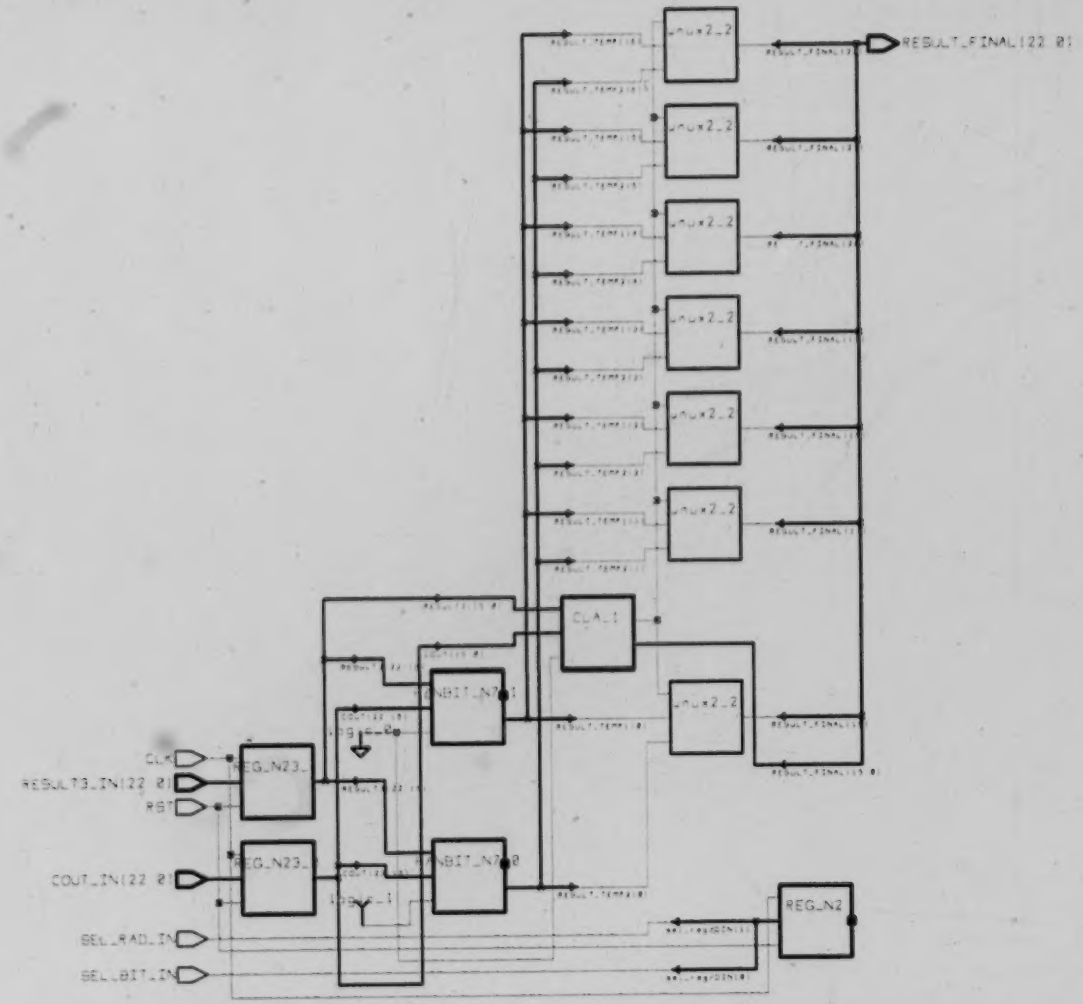
design: REG_N16	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



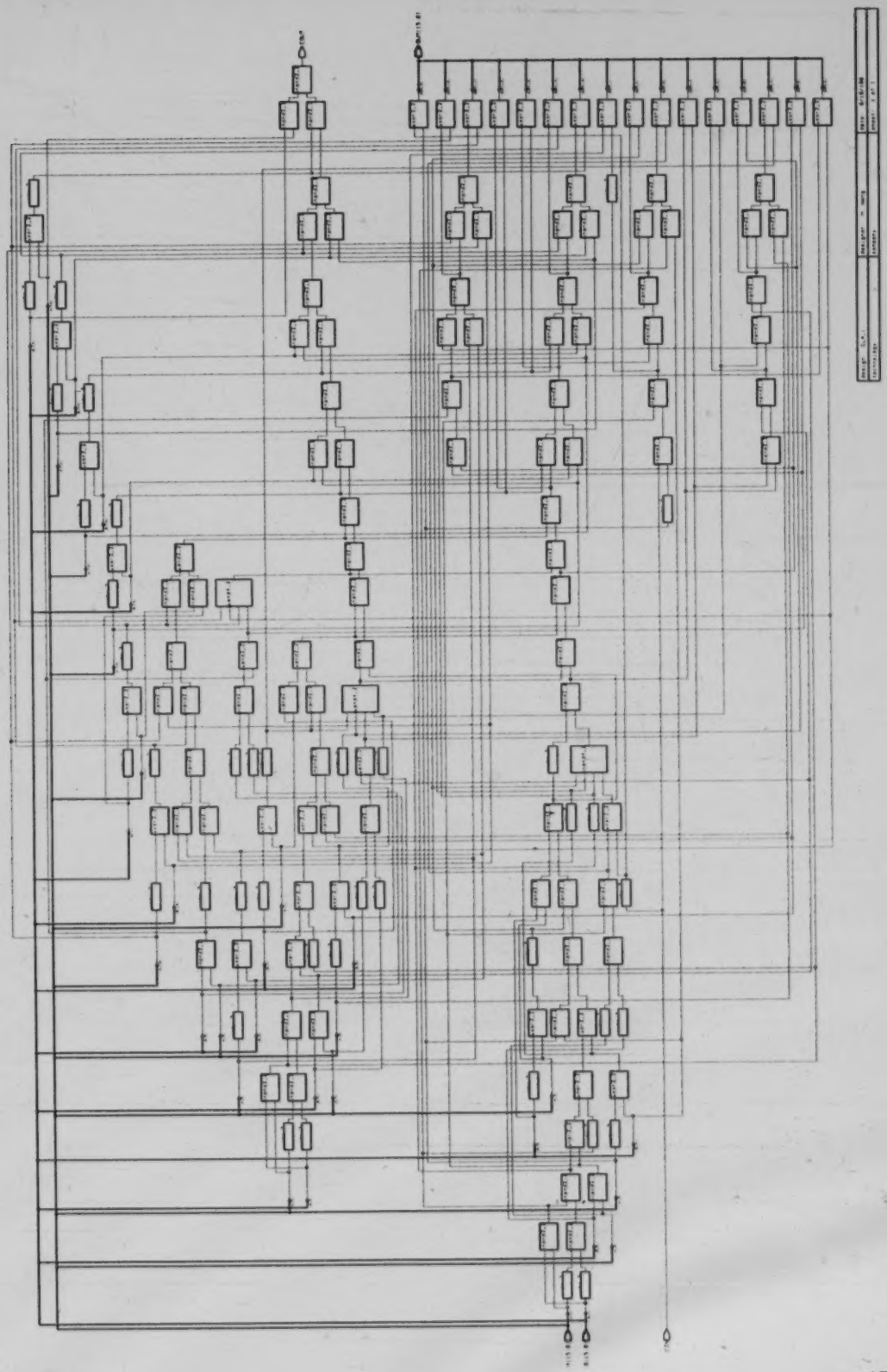
design REG_N18_8	designer M Wang	date 8/16/100
technology	company	sheet 1 of 1

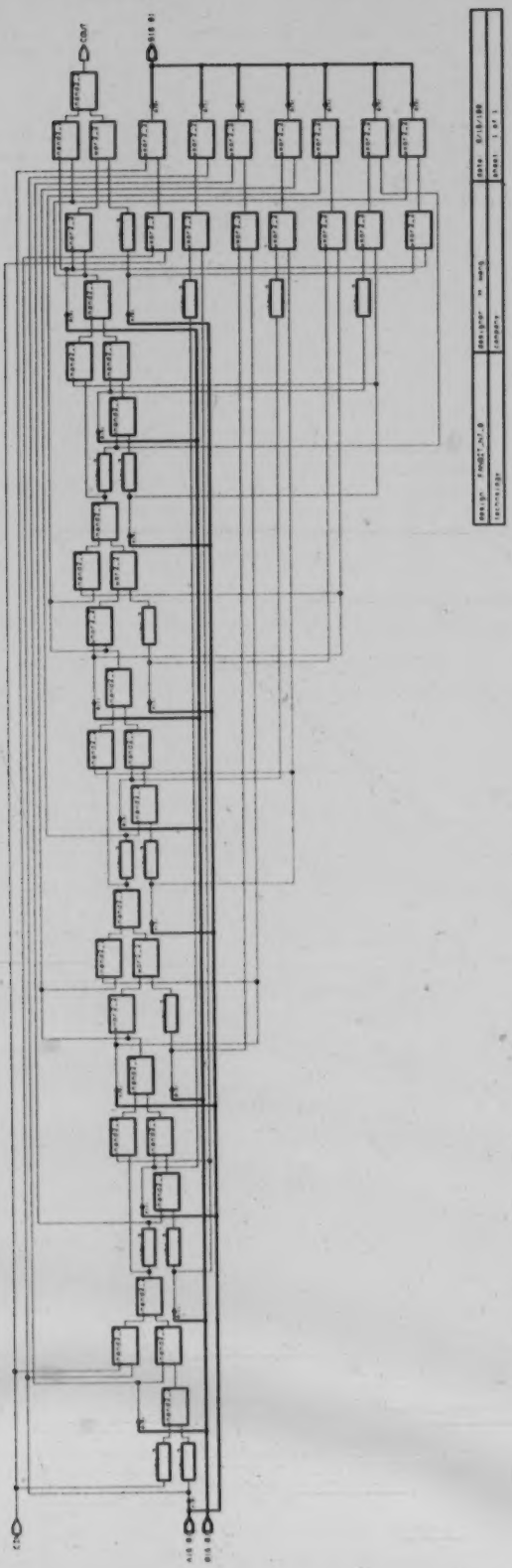
Appendix 18: Synthesized Schematic of Second Carry Propagate Addition Stage (ADD2_2)

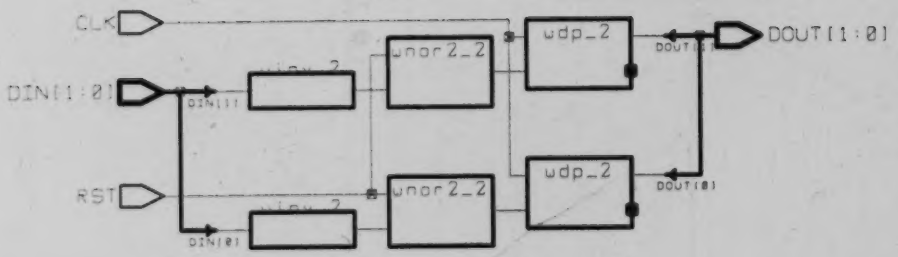
RESULT3_IN[22:0]:	*partial sum*
CLK:	*clock signal*
RST:	*reset signal*
COUT_IN[22:0]:	*carry-out*
SEL_RAD_IN:	*control signal for selecting radix-4 or radix-8 recoding scheme*
SEL_BIT_IN:	*control signal for selecting 8-bit or 12-bit operand wordlength*
RESULT_FINAL[22:0]:	*final result*
CLA_1:	*carry lookahead adder*
FANBIT_N7_i:	*7-bit full adder, i = 0, 1*
REG_N2:	*2-bit register*
REG_N23_i	*23-bit register, i = 0, 1*



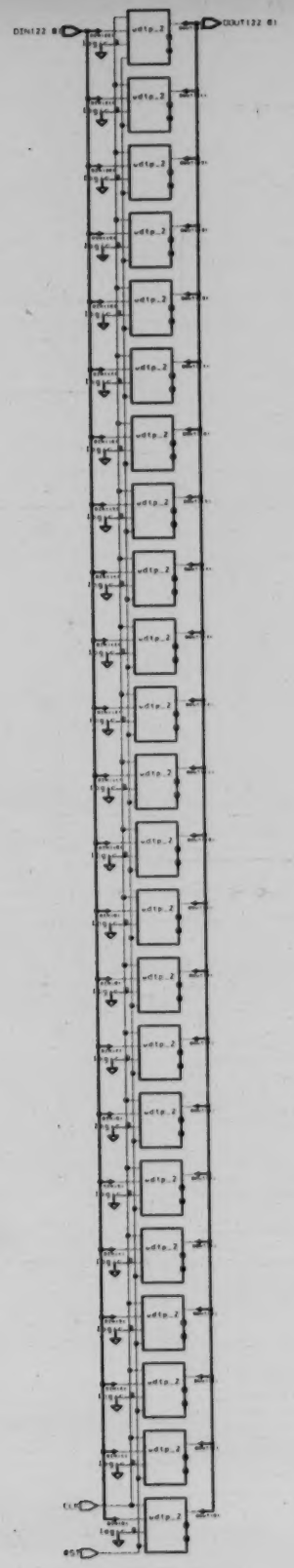
design ADD2_2	designer H. Wang	date 8/18/100
technology	company	sheet 1 of 1







design: REG_N2	designer: H. Wang	date: 8/16/100
technology:	company:	sheet: 1 of 1



design: REG_N23_0	designer: H. Wang	date: 9/18/1998
technology:	company:	sheet: 1 of 1